Acceptor and Connector

A Family of Object Creational Patterns for Initializing Communication 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 a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

1 Introduction

This paper describes the Connector and Acceptor patterns. The intent of these patterns is to decouple the active and passive initialization roles, respectively, from the tasks a communication service performs once initialization is complete. Common examples of communication services that utilize these patterns include WWW browsers, WWW servers, object request brokers, and "superservers" (which provide services like remote login and file transfer to client applications).

This paper illustrates how the Connector and Acceptor patterns can help decouple service initialization-related processing from service processing, which yields more reusable, extensible, and efficient communication software. When used in conjunction with related patterns like the Reactor [1], Active Object [2], and Service Configurator [3], the Acceptor and Connector patterns enable the creation of extensible and efficient communication software frameworks [4] and applications [5].

This paper is organized as follows: Section 2 describes the Acceptor and Connector patterns in detail; Section 3 presents concluding remarks, and the Appendix outlines background information on networking and communication protocols necessary to understand the patterns in this paper.

2 The Acceptor and Connector Patterns

2.1 Intent

The intent of these patterns is to decouple service initialization from the tasks performed once a service is initialized. The Connector pattern is responsible for *active* initialization, whereas the Acceptor pattern is responsible for *passive* initialization.

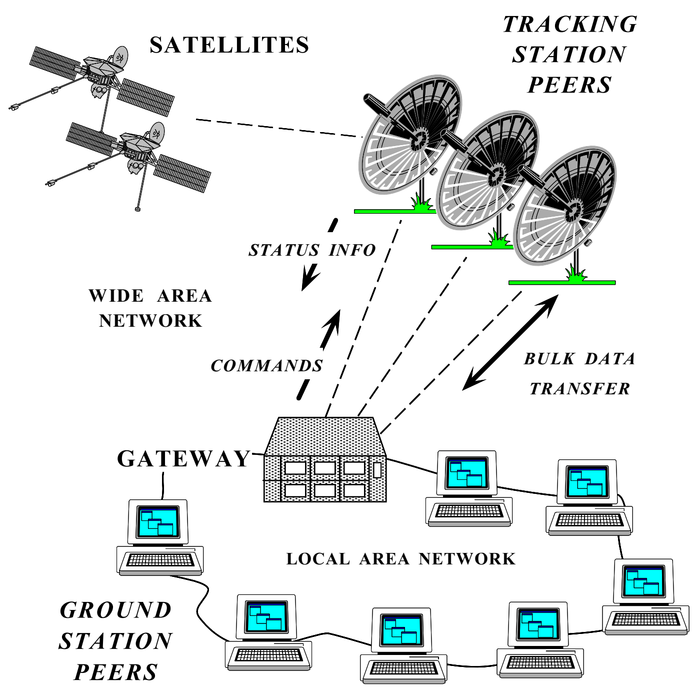

Figure 1: The Physical Architecture of a Connection-oriented Application-level Gateway

2.2 Also Known As

The Acceptor pattern is also known as the Listener [6].

2.3 Motivation

2.3.1 Context

To illustrate the Acceptor and Connector patterns, consider the multi-service, application-level `Gateway` shown in Figure 1. The `Gateway` routes several types of data (such as status information, bulk data, and commands) that are exchanged between services running on the `Peers`. The `Peers` are used to monitor and control a satellite constellation. `Peers` can be distributed throughout local area networks (LANs) and wide-area networks (WANs).

The Gateway is a Mediator [7] 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-latency WANs.

Each communication service in the Peers sends and receives status information, bulk data, and commands to and from the Gateway using separate TCP connections. Each connection is bound to a unique address (*e.g.,* an IP address and port number). For 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 network connections fail.

2.3.2 Common Traps and Pitfalls

One way to design the Peers and Gateway is to designate the initialization roles *a priori* and hard-code them into the server implementation. 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 connections and initialize the associated services.

In addition, 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 and the associated service handler objects were initialized, the Gateway could route data for each type of service it provided.

However, this approach has the following drawbacks:

• **Limited extensibility and reuse of the Gateway and Peer software:** The type of routing service (*e.g.,* status information, bulk data, or commands) performed by the Gateway is essentially independent of the mechanisms used to establish connections and initialize services. However, the hard-coded approach described above tightly couples service initialization and service behavior. This makes it hard to reuse existing services or to extend the Gateway by adding new routing services and enhancing existing services.

• **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 Peer endsystems.

• **Error-prone network programming interfaces:** Conventional network programming interfaces (such as sockets or TLI) do not provide adequate type-checking since they utilize low-level I/O handles [8]. The tight coupling of the hard-coded approach described above makes it easy to accidentally misuse these interfaces and I/O handles in ways that cannot be detected until run-time.

2.3.3 Solution

A more flexible and efficient way to design the Peers and Gateway is to use the *Acceptor* and *Connector* patterns. These two patterns decouple the *active* and *passive* initialization roles, respectively, from the communication services performed once services are initialized. These patterns resolve the following forces for communication services that use connection-oriented transport protocols:

1. The need to avoid rewriting initialization code for each new service: The Connector and Acceptor patterns permit key characteristics of services (such as application-level communication protocols and message formats) to evolve independently of the strategies used to initialize the services. Application-level service characteristics often change more frequently than initialization strategies. Therefore, this separation of concerns helps reduce software coupling and increases code reuse.

2. The need to enable flexible strategies for executing communication services concurrently: Once a connection is established and the service is initialized using the Acceptor and Connector patterns, peer applications use the connection to exchange data while performing the service. However, regardless of how the service was initialized, these services may be executed in a single-thread, in multiple threads, or multiple processes.

3. The need to make connection establishment software portable across platforms: Many operating systems provide network programming interfaces (such as sockets and TLI) and communication protocols (such as TCP/IP and IPX/SPX) whose semantics are only superficially different. Therefore, the syntactic incompatibilities of these interfaces make it hard to write portable programs, even though the initialization strategies transcend these differences. It is particularly hard to write portable asynchronous connection establishment software since asynchrony is not supported uniformly by standard network programming interfaces like sockets, CORBA, or DCOM.

4. The need to actively establish connections with large number of peers efficiently: The Connector pattern can employ asynchrony to initiate and complete multiple connections without blocking the caller. By using asynchrony, the Connector pattern enables applications to actively establish connections with a large number of peers efficiently over long-latency WANs.

5. The need to ensure that passive-mode I/O handles are not accidentally used to read or write data: Strongly decoupling the initialization role of the Acceptor pattern from the communication role of the initialized service ensures that passive-mode listener endpoints are not accidentally used incorrectly. Without this strong decoupling, services may mistakenly read or write data on passive-mode listener endpoints (which should only be used to accept connections).

As outlined above, the Connector and Acceptor patterns address very similar forces. For instance, forces 1, 2, and 3 above are resolved by both the Acceptor and Connector pattern – only the passive and active roles are reversed. Forces 4 and 5 are only resolved by one pattern each, however, due to the asymmetrical connection roles played by each pattern. For example, the Connector pattern addresses an additional force (connection scalability) by using asynchrony to actively initialize a large number of peers efficiently. This force is not addressed by the Acceptor since it is always the passive target of active initialization requests. Conversely, a Connector does not wait passively for services to initialize it. Unlike the Acceptor pattern, therefore, the Connector pattern need not decouple the listener endpoint from the data endpoint.

The following section describes the Acceptor and Connector patterns using a modified version of the GoF pattern form [7].

2.4 Applicability

- Use the Acceptor and Connector patterns when tasks performed by a service can be decoupled from the steps required to initialize the service; and
- Use the Connector pattern when an application must establish a large number of connections with peers residing across long-latency networks (such as satellite WANs); or
- Use the Acceptor pattern when connections may arrive concurrently from different peers, but blocking or continuous polling for incoming connections on any individual peer is inefficient.

2.5 Structure and Participants

The structure of the participants in the Acceptor and Connector patterns is illustrated by the Booch class diagram [9] in Figure 2 and Figure 3, respectively.[1] The two participants (`Reactor` and `Service Handler`) common to each pattern are described first, followed by participants that are unique to the Acceptor and Connector patterns:

• **Reactor:** For the Acceptor pattern, the `Reactor` demultiplexes connection requests received on one or more communication endpoints to the appropriate `Acceptor` (described

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

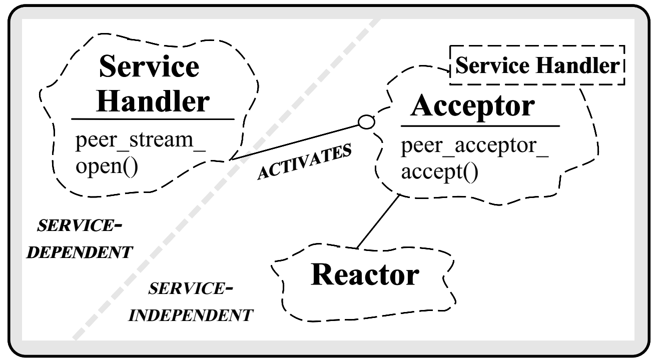

Figure 2: Structure of Participants in the Acceptor Pattern

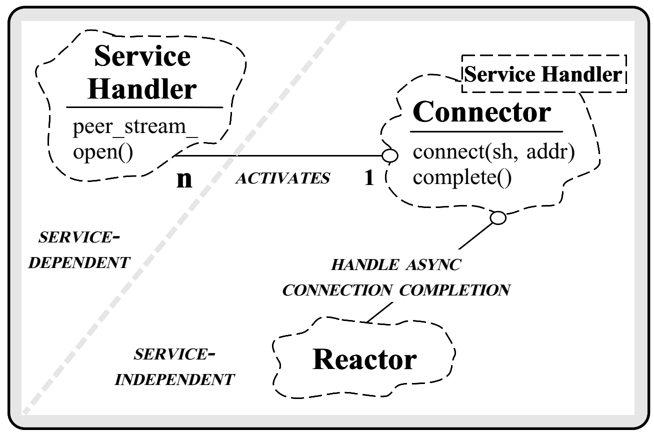

Figure 3: Structure of Participants in the Connector Pattern

below). The `Reactor` allows multiple `Acceptors` to listen for connections from different peers efficiently within a single thread of control. For the Connector pattern, the `Reactor` handles the completion of connections that were initialized asynchronously. The `Reactor` allows multiple `Service Handlers` to have their connections initiated and completed asynchronously by a `Connector` configured within a single thread of control.

• **Service Handler:** This class defines a generic interface for a service. The `Service Handler` contains a communication endpoint (`peer_stream_`) that encapsulates an I/O handle (also known as an "I/O descriptor"). This endpoint is initialized by the `Acceptor` and `Connector` and is subsequently used by the `Service Handler` to exchange data with its connected peer. The `Acceptor` and `Connector` activate a `Service Handler` by calling its `open` hook when a connection is established. Once a `Service Handler` is completely initialized (by either an `Acceptor` or a `Connector`), it typically does not interact with its initializer.

The following participant is unique to the Acceptor pattern:

• **Acceptor:** This class implements the strategy for passively initializing a `Service Handler`, which communi-

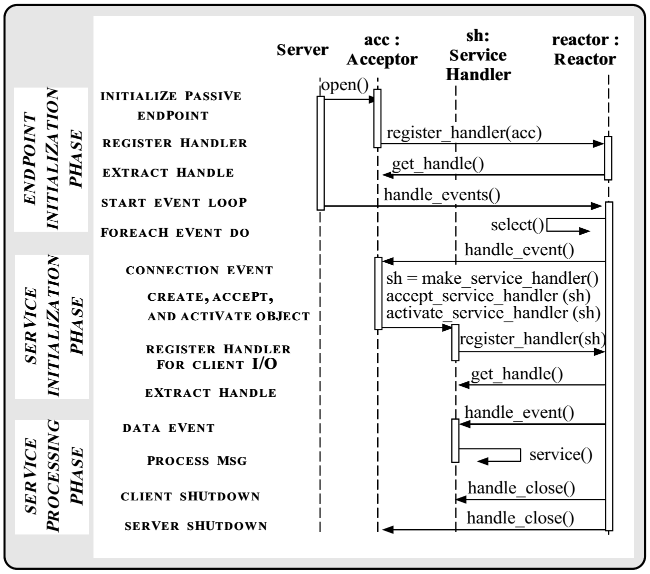

Figure 4: Collaborations Among Participants in the Acceptor Pattern

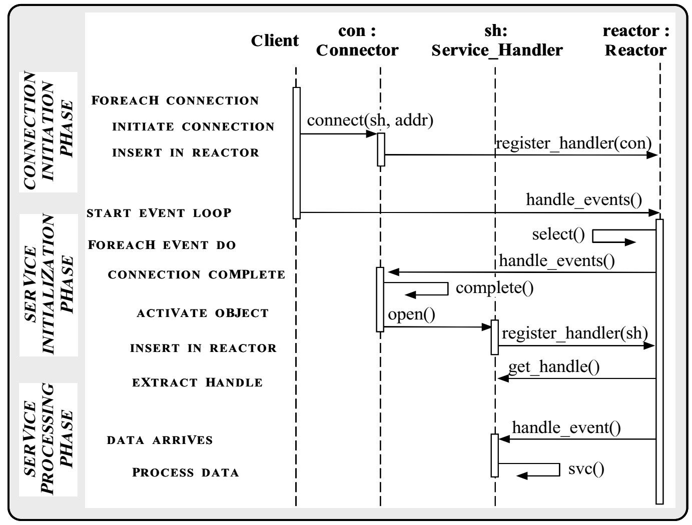

Figure 5: Collaborations Among the Connector Pattern Participants for Asynchronous Initialization

cates with the peer that actively initiated the connection. The `Reactor` calls back to the `Acceptor`'s `accept` method when a connection arrives on the passive-mode `peer_acceptor_` endpoint. The `accept` method uses this passive-mode endpoint to accept connections into the `Service Handler`'s `peer_stream_` and then activate a `Service Handler` by calling its `open` hook.

The following participant is unique to the Connector pattern:

• **Connector:** This class connects and activates a `Service Handler`. The `connect` method of a `Connector` implements the strategy for actively initializing a `Service Handler`, which communicates with the peer that passively accepts the connection. The `Connector` activates a connected `Service Handler` by calling its `open` method when initialization is complete. The `complete` method finishes activating `Service Handlers` whose connections were initiated and completed asynchronously. In this case, the `Reactor` calls back the `complete` method automatically when an asynchronous connection is established.

2.6 Collaborations

The following section describes the collaborations between participants in the Acceptor and Connector patterns.

2.6.1 Acceptor 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 the `Service Handler` associated with the passive-mode endpoint. When a connection arrives, the `Reactor` calls back to the `Acceptor`'s `accept` method. This method performs the strategy for initializing a `Service Handler`. The `Acceptor`'s strategy assembles the resources necessary to (1) create a new `Concrete Service Handler` object, (2) accept the connection into this object, and (3) activate the `Service Handler` by calling its `open` hook. The `open` hook of the `Service Handler` then performs service-specific initialization (such as allocating locks, opening log files, etc.).

3. Service processing phase: Once the connection has been established passively and the service has been initialized, service processing begins. In this phase, application-specific tasks process the data exchanged between the `Service Handler` and its connected `Peer`.

2.6.2 Connector Collaborations

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

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

2. Service initialization phase: which activates the `Service Handler` by calling its `open` hook when the connection completes successfully. The `open` hook of the `Service Handler` then performs service-specific initialization.

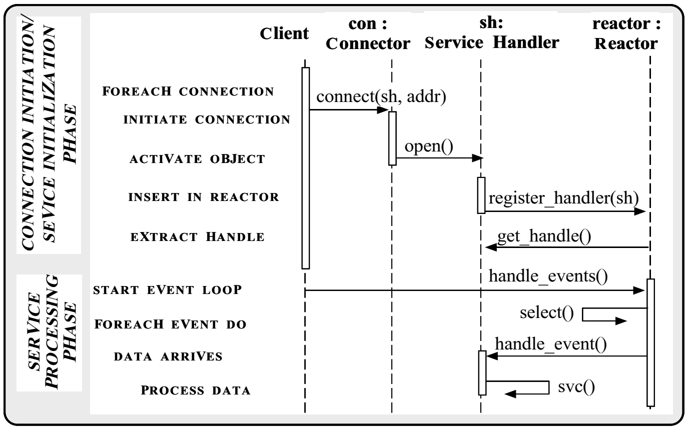

Figure 6: Collaborations Among the Connector Pattern Participants for Synchronous Initialization

3. Service processing phase: once the `Service Handler` is activated, it performs the application-specific service processing using the data exchanged with its connected `Peer`.

Figure 5 illustrates these three phases of collaboration using *asynchronous* service initialization. Note how the connection initiation phase is temporally separated from the service initialization phase. This enables multiple connection initiations and completions to proceed in parallel within each thread of control.

The collaboration for *synchronous* service initialization is shown in Figure 6. In this case, the `Connector` combines the connection initiation and service initialization phases into a single blocking operation. Note, however, that only one connection is established per-thread for each invocation of `connect`.

In general, synchronous service initialization is useful for the following situations:

- If the latency for establishing a connection is very low (*e.g.*, establishing a connection with a server on the same host via the loopback device); or
- If multiple threads of control are available and it is feasible to use a different thread to connect each `Service Handler` synchronously; or
- If the services must be initialized in a fixed order and clients cannot perform useful work until a connection is established.

In contrast, asynchronous service initialization is useful for the following situations:

- If the connection latency is high and there are many peers to connect with (*e.g.*, establishing a large number of connections over a high-latency WAN); or
- If only a single thread of control is available (*e.g.*, if the OS platform does not provide application-level threads); or
- If the order in which services are initialized is not important and if the client application must perform additional work (such as refreshing a GUI) while the connection is in the process of being established.

2.7 Consequences

2.7.1 Benefits

The Acceptor and Connector patterns provide the following benefits:

• **Enhances the reusability, portability, and extensibility of connection-oriented software** by decoupling mechanisms for passively initializing services from the tasks performed by the services. For instance, the application-independent mechanisms in the `Acceptor` and `Connector` are reusable components that know how to (1) establish connections passively and (2) initialize the associated `Service Handler`. In contrast, the `Service Handler` knows how to perform application-specific service processing.

This separation of concerns is achieved by decoupling the initialization strategy from the service handling strategy. Thus, each strategy can evolve independently. The strategy for active initialization can be written once, placed into a class library or framework, and reused via inheritance, object composition, or template instantiation. Thus, the same passive initialization code need not be rewritten for each application. Services, in contrast, may vary according to different application requirements. By parameterizing the `Acceptor` and `Connector` with a `Service Handler`, the impact of this variation is localized to a single point in the software.

• **Improves application robustness:** Application robustness is improved by strongly decoupling the `Service Handler` from the `Acceptor`. This decoupling ensures that the passive-mode `peer_acceptor_` cannot accidentally be used to read or write data. This eliminates a common class of errors that can arise when programming with weakly typed network programming interfaces such as sockets or TLI [8].

• **Efficiently utilize the inherent parallelism in the network and hosts:** By using the asynchronous mechanisms shown in Figure 5, the Connector pattern can actively establish connections with a large number of peers efficiently over long-latency WANs. This is an important property since a large distributed system may have several hundred `Peers` connected to a single `Gateway`. One way to connect all these `Peers` to the `Gateway` is to use the synchronous mechanisms shown in Figure 6. However, the round trip delay for a 3-way TCP connection handshake over a long-latency WAN (such as a geosynchronous satellite or trans-atlantic fiber cable) may take several seconds per handshake. In this case, synchronous connection mechanisms cause unnecessary delays since the inherent parallelism of the network and computers is underutilized.

2.7.2 Drawbacks

The Acceptor and Connector patterns have the following drawbacks:

• **Additional indirection:** Both the Acceptor and Connector patterns may require additional indirection compared with using the underlying network programming interfaces directly. However, languages that support parameterized types (such as C++, Ada, or Eiffel), can often implement these patterns with no significant overhead since compilers can inline the method calls used to implement these patterns.

• **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.

2.8 Implementation

This section describes how to implement the Acceptor and Connector patterns in C++. The implementation described below is based on reusable components provided in the ACE OO network programming toolkit [4].

Figure 7 divides participants in the Acceptor and Connector patterns into the *Reactive*, *Connection*, and *Application* layers.[2] The Reactive and Connection layers perform generic, application-independent strategies for handling events and initializing services, 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 in this implementation of the Acceptor and Connector patterns.

The implementations of the Acceptor and Connector patterns are structured very similarly. The Reactive layer is identical in both, and the roles of the `Service Handler` and `Concrete Service Handler` are also similar. Moreover, the `Acceptor` and `Concrete Acceptor` play roles equivalent to the `Connector` and `Concrete Connector` classes. The primary difference between the two patterns is that in the Acceptor pattern these two classes play a *passive* role in establishing a connection. In contrast, in the Connector pattern they play an *active* role.

2.8.1 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 in 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` [10], 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. The implementation of the Reactor pattern is described in [1]. The two main roles in the Reactive layer are summarized below:

• **Reactor:** This class defines an interface for registering, removing, and dispatching `Event Handler` objects (such as the `Acceptor`, `Connector`, and `Service 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, a completion of a connection request started asynchronously, or the arrival of data from a connected peer.

2.8.2 Connection Layer

The Connection layer is responsible for (1) creating a `Service Handler`, (2) passively or actively connecting it with a peer, and (3) activating it once it is connected. Since all behavior in this layer is completely generic, these classes delegate to the concrete IPC mechanism and `Concrete Service Handler` instantiated by the Application layer (described below). Likewise, the Connection layer delegates to the `Reactor` to handle initialization-related events (such as establishing connections asynchronously without requiring multi-threading). The three primary roles (*i.e.,* `Service Handler`, `Acceptor`, and `Connector`) in the Connection layer are described below.

• **Service Handler:** This abstract class provides a generic interface for processing services. Applications must customize this class to perform a particular type of service. The middle part of Figure 7 illustrates the interface of the `Service Handler`. The interface of the `Service Handler` is shown below:

```
// PEER_STREAM is the type of the
// Concrete IPC mechanism.
template <class PEER_STREAM>
class Service_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 peer_stream_; }

protected:
  // Concrete IPC mechanism instance.
  PEER_STREAM peer_stream_;
};
```

[2] 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.

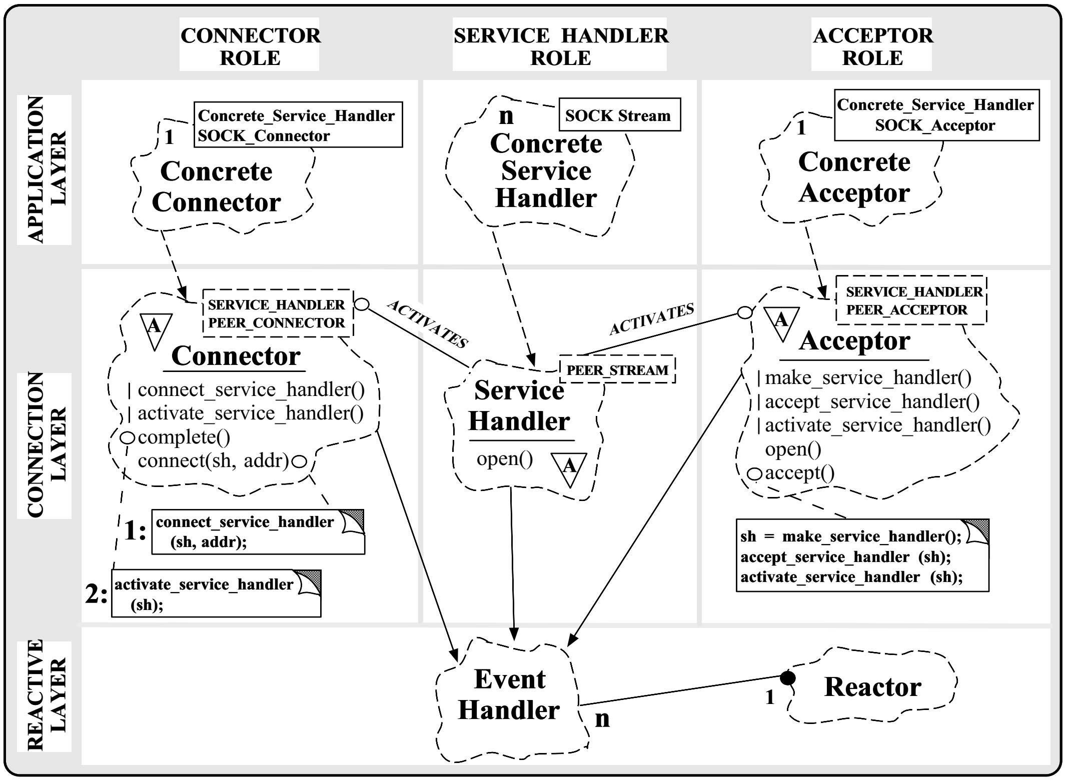

Figure 7: Layering and Partitioning of Participants in the Acceptor and Connector Patterns

The `open` hook of a `Service Handler` is called by the `Acceptor` or `Connector` once a connection is established. The behavior of this pure virtual method must be defined by a subclass, which typically performs any service-specific initializations.

`Service Handler` subclasses can also define the service's concurrency strategy. For example, a `Service Handler` may inherit from the `Event Handler` and employ the Reactor [1] pattern to process data from peers in a single-thread of control. Conversely, a `Service Handler` might use the Active Object pattern [2] to process incoming data in a different thread of control than the one the `Acceptor` object used to connect it. Section 2.9 illustrates how several different concurrency strategies can be configured flexibly without affecting the structure or behavior of the Acceptor or Connector patterns.

• **Connector:** This abstract class implements the generic strategy for actively initializing communication services. The left part of Figure 7 illustrates the interface of the `Connector`. The key methods and objects in the `Connector` are shown below:

```
// The SERVICE_HANDLER is the type of service.
// The PEER_CONNECTOR is the type of concrete
// IPC active connection mechanism.
template <class SERVICE_HANDLER,
          class PEER_CONNECTOR>
class Connector : public Event_Handler
{
public:
  enum Connect_Mode {
    SYNC, // Initiate connection synchronously.
    ASYNC // Initiate connection asynchronously.
  };

  // Initialization method.
  Connector (void);

  // Actively connecting and activate a service.
  int connect (SERVICE_HANDLER *sh,
               const PEER_CONNECTOR::PEER_ADDR &addr,
               Connect_Mode mode);

protected:
  // Defines the active connection strategy.
  virtual int connect_service_handler
    (SERVICE_HANDLER *sh,
     const PEER_CONNECTOR::PEER_ADDR &addr,
     Connect_Mode mode);

  // Register the SERVICE_HANDLER so that it can
  // be activated when the connection completes.
  int register_handler (SERVICE_HANDLER *sh,
                        Connect_Mode mode);

  // Defines the handler's concurrency strategy.
  virtual int activate_service_handler
    (SERVICE_HANDLER *sh);

  // Activate a SERVICE_HANDLER whose
  // non-blocking connection completed.
  virtual int complete (HANDLE handle);

private:
  // IPC mechanism that establishes
  // connections actively.
  PEER_CONNECTOR connector_;

  // Collection that maps HANDLEs
  // to SERVICE_HANDLER *s.
```

```
  Map_Manager<HANDLE, SERVICE_HANDLER *>
    handler_map_;

  // Inherited from the Event_Handler -- will be
  // called back by Eactor when events complete
  // asynchronously.
  virtual int handle_event (HANDLE, EVENT_TYPE);
};

// Useful "short-hand" macros used below.
#define SH SERVICE_HANDLER
#define PC PEER_CONNECTION
```

The `Connector` is parameterized by a particular type of `PEER CONNECTOR` and `SERVICE HANDLER`. The `PEER CONNECTOR` provides the transport mechanism used by the `Connector` to actively establish the connection synchronously or asynchronously. The `SERVICE 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 acceptance protocol.

The use of parameterized types is an implementation decision that helps improve portability by allowing the wholesale replacement of the mechanisms used by the Connector. 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 CONNECTOR` template argument can be instantiated with either a `SOCK Connector` or a `TLI Connector`, 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 [7]. Parameterized types improve run-time efficiency at the expense of additional space and time overhead during program compiling and linking.

The implementation of the `Connector`'s `connect` method is outlined in Figure 7.[3] The `connect` method is public entry point for a `Connector`, as shown below.

```
template <class SH, class PC> int
Connector<SH, PC>::connect
     (SERVICE_HANDLER *service_handler,
      const PEER_CONNECTOR::PEER_ADDR &addr,
      Connect_Mode mode)
{
  connect_service_handler (service_handler,
                           addr, mode);
}
```

This method provides the external entry point into the `Connector` factory. It uses the Bridge pattern[4] to delegate to the `Connector`'s connection strategy, `connect_service_handler`, which initiates a connection:

[3] To save space, most of the error handling in this paper has been omitted.

[4] The use of the Bridge pattern allows subclasses of `Connector` to transparently modify the connection strategy, without changing the interface.

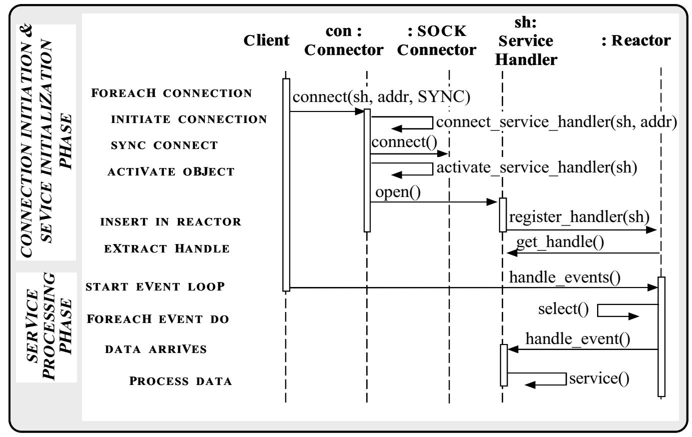

Figure 8: Collaborations Among the Connector Participants for Synchronous Initialization

```
template <class SH, class PC> int
Connector<SH, PC>::connect_service_handler
     (SERVICE_HANDLER *service_handler,
      const PEER_CONNECTOR::PEER_ADDR &remote_addr,
      Connect_Mode mode)
{
  // Delegate to concrete PEER_CONNECTOR
  // to establish the connection.

  if (connector_.connect (*service_handler,
                          remote_addr,
                          mode) == -1) {
    if (mode == ASYNC && errno == EWOULDBLOCK)
      // If the connection hasn't completed and
      // we are using non-blocking semantics then
      // register ourselves with the Reactor
      // Singleton so that it will callback when
      // the connection is complete.
      Reactor::instance ()->register_handler
                               (this, WRITE_MASK);

      // Store the SERVICE_HANDLER in the map of
      // pending connections.
      handler_map_.bind
        (connector_.get_handle (), service_handler);
  }
  else if (mode == SYNC)
    // Activate if we connect synchronously.
    activate_service_handler (service_handler);
}
```

If the value of the `Connect_Mode` parameter is `SYNC` the `SERVICE HANDLER` will be activated after the connection completes synchronously, as illustrated in Figure 8.

To connect with multiple `Peers` efficiently, the `Connector` must be able to actively establish connections asynchronously, *i.e.*, without blocking the caller. Asynchronous behavior is specified by passing the `ASYNC` connection mode to `Connector::connect`, as illustrated in Figure 9.

Once instantiated, the `PEER CONNECTOR` class provides the concrete IPC mechanism for initiating connections asynchronously. The implementation of the Connector pattern shown here uses asynchronous I/O mechanisms provided by the operating system and communication protocol stack (*e.g.*, by setting sockets into non-blocking

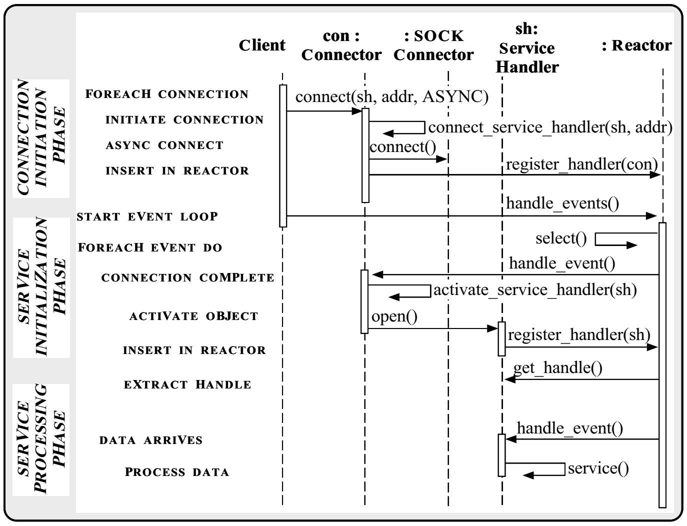

Figure 9: Collaborations Among the Connector Participants for Asynchronous Initialization

mode and using an event demultiplexer like `select` or `WaitForMultipleObjects` to determine when the I/O completes).

The `Connector` maintains a map of `Service Handlers` whose asynchronous connections are pending completion. Since the `Connector` inherits from `Event Handler`, the `Reactor` can automatically call back to the `Connector's handle_event` method when a connection completes.

The `handle_event` method is an Adapter that transforms the `Reactor`'s event handling interface to a call to the Connector pattern's `complete` method, which activates by `SERVICE HANDLER` by invoking its `open` hook. The `open` hook is called when a connection is established successfully, regardless of whether connections are established synchronously or asynchronously. This uniformity of behavior makes it possible to write services whose behavior can be decoupled from the manner by which they are actively connected and initialized.

The `Connector's handle_event` method is shown below:

```
template <class SH, class PC> int
Connector<SH, PC>::handle_event (HANDLE handle,
                                 EVENT_TYPE type)
{
  // Adapt the Reactor's event handling API to
  // the Connector's API.
  complete (handle);
}
```

The `complete` method then activates a `SERVICE HANDLER` whose non-blocking connection just completed successfully:

```
template <class SH, class PC> int
Connector<SH, PC>::complete (HANDLE handle)
{
  SERVICE_HANDLER *service_handler = 0;

  // Locate the SERVICE_HANDLER corresponding
  // to the HANDLE.
  handler_map_.find (handle, service_handler);

  // Transfer I/O handle to SERVICE_HANDLER *.
  service_handler->set_handle (handle);

  // Remove handle from Reactor.
  Reactor::instance ()->remove_handler
                        (handle, WRITE_MASK);

  // Remove handle from the map.
  handler_map_.unbind (handle);

  // Connection is complete, so activate handler.
  activate_service_handler (service_handler);
}
```

The `complete` method finds and removes the connected `SERVICE HANDLER` from its internal map, transfers the I/O `HANDLE` to the `SERVICE HANDLER`, and initializes the service by calling `activate_service_handler`. This method delegates to the concurrency strategy designated by the `SERVICE HANDLER::open` hook, as follows:

```
template <class SH, class PC> int
Connector<SH, PC>::activate_service_handler
  (SERVICE_HANDLER *service_handler)
{
  service_handler->open ();
}
```

- **Acceptor:** This abstract class implements the generic strategy for passively initializing communication services. The right-hand part of Figure 7 illustrates the interface of the `Acceptor`. The key methods and objects in the `Acceptor` are shown below:

```
// The SERVICE_HANDLER is the type of service.
// The PEER_ACCEPTOR is the type of concrete
// IPC passive connection mechanism.
template <class SERVICE_HANDLER,
          class PEER_ACCEPTOR>
class Acceptor : public Event_Handler
{
public:
  // Initialize local_addr listener endpoint
  // and register with Reactor Singleton.
  virtual int open
    (const PEER_ACCEPTOR::PEER_ADDR &local_addr);

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

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

  // Defines the handler's connection strategy.
  virtual int accept_service_handler
              (SERVICE_HANDLER *);

  // Defines the handler's concurrency strategy.
  virtual int activate_service_handler
              (SERVICE_HANDLER *);

  // Demultiplexing hooks inherited from
  // Event_Handler -- used by Reactor for callbacks.
  virtual HANDLE get_handle (void) const;
  virtual int handle_close (void);
```

```
  // Invoked when connection requests arrive.
  virtual int handle_event (HANDLE, EVENT_TYPE);

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

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

The Acceptor is parameterized by a particular type of PEER ACCEPTOR and SERVICE HANDLER. The PEER ACCEPTOR provides the transport mechanism used by the Acceptor to passively establish the connection. The SERVICE 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.

As with the Connector, 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. The implementation of the Acceptor's methods is presented below.

Applications use the open hook 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)
{
  // Forward initialization to the PEER_ACCEPTOR.
  peer_acceptor_.open (local_addr);

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

The open hook 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 [10], TLI [6], STREAM pipes [12], Win32 Named Pipes, etc.

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

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

The Reactor stores this HANDLE internally and uses it to detect and demultiplex incoming connection from clients. Since the Acceptor class inherits from Event Handler, the Reactor can automatically call back to the Acceptor's handle_event method when a connection arrives from a peer. This method is an Adapter that transforms the Reactor's event handling interface to a call to the Acceptor's accept method, as follows:

```
template <class SH, class PA> int
Acceptor<SH, PA>::handle_event (HANDLE,
                                EVENT_TYPE)
{
  // Adapt the Reactor's event handling API to
  // the Acceptor's API.
  accept ();
}
```

As shown below, the accept method is a Template Method [7] that implements the Acceptor pattern's passive initialization strategy for creating a new SERVICE HANDLER, accepting a connection into it, and activating the service:

```
template <class SH, class PA> int
Acceptor<SH, PA>::accept (void)
{
  // Create a new SERVICE_HANDLER.
  SH *service_handler = make_service_handler ();

  // Accept connection from client.
  accept_service_handler (service_handler);

  // Activate SERVICE_HANDLER by calling
  // its open() hook.
  activate_service_handler (service_handler);
}
```

This method is very concise since it factors all low-level details into the concrete SERVICE HANDLER and PEER ACCEPTOR instantiated via 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. This flexibility makes it possible to write services whose behavior can be decoupled from the manner by which they are passively connected and initialized.

The Acceptor's default strategy for creating SERVICE HANDLERs is defined by the make_service_handler method:

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

The default behavior uses a "demand strategy," which creates a new SERVICE HANDLER for every new connection. However, subclasses of Acceptor can override this strategy to create SERVICE HANDLERs using other strategies (such as creating an individual Singleton [7] or dynamically linking the `SERVICE HANDLER` from a shared library).

The `SERVICE HANDLER` connection acceptance strategy used by the `Acceptor` is defined below by the `accept_service_handler` method:

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

The default behavior delegates to the `accept` method provided by the `PEER ACCEPTOR`. Subclasses can override the `accept_service_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 SERVICE HANDLER` concurrency strategy is defined by the `activate_service_handler` method:

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

The default behavior of this method is to activate the `SERVICE HANDLER` by calling its `open` hook. This allows the `SERVICE HANDLER` to select its own concurrency strategy. For instance, if the `SERVICE HANDLER` inherits from `Event Handler` it can register with the `Reactor`. This allows the `Reactor` to dispatch the `SERVICE 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 `SERVICE HANDLER` an "active object" [2] that processes data using multi-threading or multi-processing).

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, which enables it to release any 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 peer_acceptor_.close ();
}
```

2.8.3 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 2.9 are part of the `SOCK SAP` C++ wrapper library for sockets [8]. `SOCK SAP` encapsulates the stream-oriented semantics of connection-oriented protocols like TCP and SPX with efficient, portable, and type-safe C++ wrappers.

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

• **Concrete Service Handler:** This class implements the concrete application-specific service activated by a `Concrete Acceptor` or a `Concrete Connector`. A `Concrete Service Handler` is instantiated with a specific type of C++ IPC wrapper that exchanges data with its connected peer. The sample code examples in Section 2.9 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 Service Handler` with a different `PEER STREAM` (such as an SVR4 `TLI Stream` or a Win32 `Named Pipe Stream`).

• **Concrete Connector:** This class instantiates the generic `Connector` factory with concrete parameterized type arguments for `SERVICE HANDLER` and `PEER CONNECTOR`.

• **Concrete Acceptor:** This class instantiates the generic `Acceptor` factory with concrete parameterized type arguments for `SERVICE HANDLER` and `PEER ACCEPTOR`.

In the sample code in Section 2.9, `SOCK Connector` and `SOCK Acceptor` are the underlying transport programming interfaces used to establish connections actively and passively, respectively. However, parameterizing the `Connector` and `Acceptor` with different mechanisms (such as a `TLI Connector` or `Named Pipe Acceptor`) is straightforward since the IPC mechanisms are encapsulated in C++ wrapper classes.

The following section illustrates sample code that instantiates a `Concrete Service Handler`, `Concrete Connector`, and `Concrete Acceptor` to implement the `Peers` and `Gateway` described in Section 2.3. This particular example of the Application layer customizes the generic initialization strategies provided by the `Connector` and `Acceptor` components in the Connection layer. Note how the use of templates and dynamic binding permits specific details (such as the underlying network programming interface or the creation strategy) to change flexibly. For instance, no `Connector` components must change when the concurrency strategy is modified in Section 2.9.1 and Section 2.9.2.

2.9 Sample Code

The sample code below illustrates how the `Peers` and `Gateway` described in Section 2.3 use the Acceptor and Connector patterns to simplify the task of passively initializing services. The `Peers` play the passive role in establishing connections with the `Gateway`, whose connections are initiated actively by using the Connector pattern. Figure 10

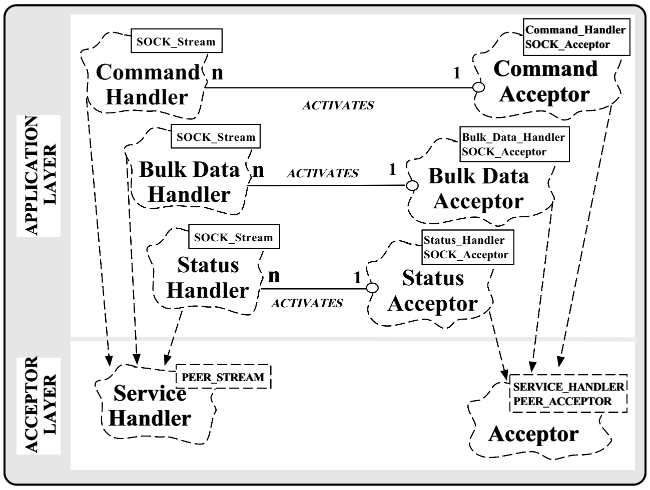

Figure 10: Structure of Acceptor Pattern Participants for `Peers`

illustrates how participants in the Acceptor pattern are structured in a `Peer` and Figure 11 illustrates how participants in the Connector pattern are structured in the `Gateway`.

2.9.1 Peer Components

• **Service Handlers for Communicating with a Gateway:** The classes shown below, `Status Handler`, `Bulk Data Handler`, and `Command Handler`, process routing messages sent and received from a `Gateway`. Since these `Concrete Service Handler` classes inherit from `Service Handler` they are capable of being passively initialized by an `Acceptor`.

To illustrate the flexibility of the Acceptor pattern, each `open` routine in the `Service Handlers` can implement 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 the `Reactor` that demultiplexes connection requests for the `Acceptor` factories. Note how changes to these concurrency strategies do not affect the implementation of the `Acceptor`, which is generic and thus highly flexible and reusable.

We start by defining a `Service Handler` that uses `SOCK Stream` for socket-based data transfer:

```
typedef Service_Handler <SOCK_Stream> PEER_HANDLER;
```

The `PEER HANDLER` typedef 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) {
```

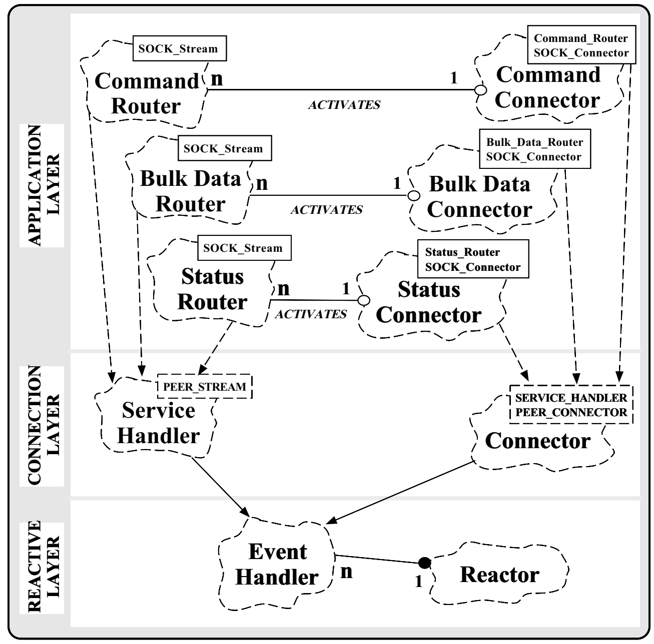

Figure 11: Structure of Connector Pattern Participants for the `Gateway`

```
    // Make handler run in separate thread (note
    // that  Thread::spawn requires a pointer to
    // a static method as the thread entry point).

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

  // Static entry point into thread, which blocks
  // on the handle_event () call in its own thread.
  static void *service_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];
    stream_.recv (buf, sizeof buf);
    // ...
  }

  // ...
};
```

The `Bulk Data Handler` and `Command Handler` classes can likewise be defined as subclasses of `PEER HANDLER`. For instance, 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 child process.
      // This method can block since it
      // runs in its own process.
      while (handle_event () != -1)
```

```
        continue;
    // ...
  }

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

  // ...
};
```

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

```
class Command_Handler : public PEER_HANDLER
{
public:
  // Performs handler activation.
  virtual int open (void) {
    // Handler runs in same thread as main
    // Reactor singleton.
    Reactor::instance ()->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.
    stream_.recv (buf, sizeof buf);
    // ...
  }

  //...
};
```

• **Acceptors for creating Peer Service 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.

```
// 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;
```

• **The Peer Main function:** The main program initializes the concrete `Acceptor` factories by calling their `open` hooks with the well-known ports for each service. As shown in Section 2.8.2, the `Acceptor::open` method registers itself with an 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 the appropriate `PEER HANDLER` to perform the service, accepts the connection into the handler, and activates the handler.

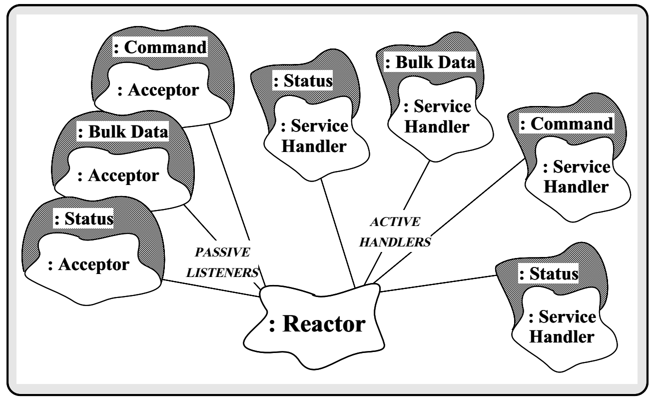

Figure 12: Object Diagram for the Acceptor Pattern in the Peer

```
// Main program for the Peer.

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

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

  for (;;)
    Reactor::instance ()->handle_events ();
}
```

Figure 12 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.

2.9.2 Gateway Components

• **Service Handlers for Gateway routing:** The classes shown below, `Status Router`, `Bulk Data Router`, and `Command Router`, route data they receive from a source `Peer` to one or more destination `Peers`. Since these `Concrete Service Handler` classes inherit from `Service Handler` they can be actively connected and initialized by a `Connector`.

To illustrate the flexibility of the Connector pattern, each `open` routine in a `Service Handler` implements a different concurrency strategy. In particular, when the `Status Router` is activated it runs in a separate thread; the `Bulk Data Router` runs as a separate process; and the `Command Router` runs in the same thread as the `Reactor` that demultiplexes connection completion events for the `Connector` factory. As with the `Acceptor`, note how changes to these concurrency strategies do not affect the implementation of the `Connector`, which is generic and thus highly flexible and reusable.

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

```
typedef Service_Handler <SOCK_Stream> PEER_ROUTER;
```

This class forms the basis for all the subsequent routing services. For instance, the `Status Router` class routes status data from/to `Peers`:

```
class Status_Router : public PEER_ROUTER
{
public:
  // Activate router in separate thread.
  virtual int open (void) {
    // Thread::spawn requires a pointer to a
    //  static method as the thread entry point).
    Thread::spawn (&Status_Router::service_run,
                   this);
  }

  // Static entry point into thread, which blocks
  // on the handle_event() call in its own thread.
  static void *service_run (Status_Router *this_) {
    // This method can block since it
    // runs in its own thread.
    while (this_->handle_event () != -1)
      continue;
  }

  // Receive and route status data from/to Peers.
  virtual int handle_event (void) {
    char buf[MAX_STATUS_DATA];
    peer_stream_.recv (buf, sizeof buf);
    // Routing takes place here...
  }

  // ...
};
```

The `Bulk Data Router` and `Command Router` classes can likewise be defined as subclasses of `PEER ROUTER`. For instance, the `Bulk Data Router` routes bulk data from/to `Peers`:

```
class Bulk_Data_Router : public PEER_ROUTER
{
public:
  // Activates router in separate process.
  virtual int open (void) {
    if (fork () == 0) // In child process.
      // This method can block since it
      // runs in its own process.
      while (handle_event () != -1)
        continue;
    // ...
  }

  // Receive and route bulk data from/to Peers.
  virtual int handle_event (void) {
    char buf[MAX_BULK_DATA];
    peer_stream_.recv (buf, sizeof buf);
    // Routing takes place here...
  }

};
```

The `Command Router` class routes Command data from/to `Peers`:

```
class Command_Router : public PEER_ROUTER
{
public:
  // Activates router in same thread as Connector.
  virtual int open (void) {
    Reactor::instance ()->register_handler
                            (this, READ_MASK);
  }

  // Receive and route command data from/to Peers.
  virtual int handle_event (void) {
    char buf[MAX_COMMAND_DATA];
    // This method cannot block since it borrows
    // the thread of control from the Reactor.
    peer_stream_.recv (buf, sizeof buf);
    // Routing takes place here...
  }
};
```

- **A Connector for creating Peer Service Handlers:** The following `typedef` defines a `Connector` factory specialized for `PEER ROUTERS`:

```
typedef Connector<PEER_ROUTERS, SOCK_Connector>
        PEER_CONNECTOR;
```

- **The Gateway Main function:** The main program for the `Gateway` is shown below. The `get_peer_addrs` function creates the `Status`, `Bulk Data`, and `Command Routers` that route messages through the `Gateway`. This function (whose implementation is not shown) reads a list of `Peer` addresses from a configuration file. Each `Peer` address consists of an IP address and a port number. Once the `Routers` are initialized, the `Connector` factories defined above initiate all the connections asynchronously (indicated by passing the `ASYNC` flag to the `connect` method).

```
// Main program for the Gateway.

// Obtain lists of Status_Routers,
// Bulk_Data_Routers, and Command_Routers
// from a config file.

void get_peer_addrs (Set<PEER_ROUTERS> &peers);

int main (void)
{
  // Connection factory for PEER_ROUTERS.
  PEER_CONNECTOR peer_connector;

  // A set of PEER_ROUTERs that perform
  // the Gateway's routing services.
  Set<PEER_ROUTER> peers;

  // Get set of Peers to connect with.
  get_peer_addrs (peers);

  // Iterate through all the Routers and
  // initiate connections asynchronously.
  PEER_ROUTER *peer;

  for (Set_Iter<PEER_ROUTER> set_iter (peers);
       set_iter.next (peer) != 0;
       set_iter++)
    peer_connector.connect (peer,
                            peer->address (),
                            PEER_CONNECTOR::ASYNC);
  // Loop forever handling connection completion
  // events and routing data from Peers.

  for (;;)
    Reactor::instance ()->handle_events ();
  /* NOTREACHED */
}
```

All connections are invoked asynchronously. They complete concurrently via `Connector::complete` method, which are called back within the `Reactor`'s event loop. The `Reactor` also demultiplexes and dispatches routing

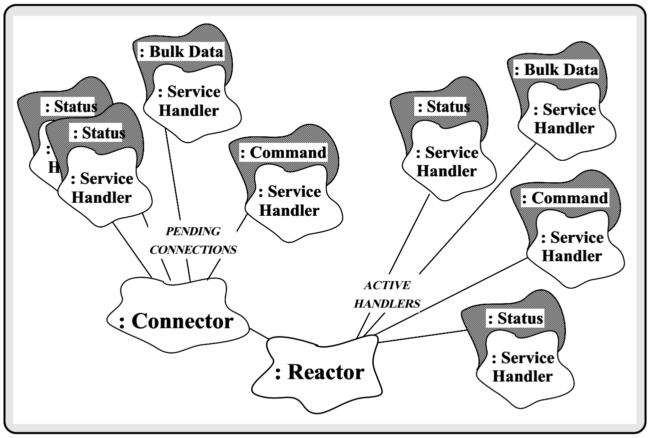

Figure 13: Object Diagram for the Connector Pattern in the Gateway

events for `Command Router` objects, which run in the `Reactor`'s thread of control. The `Status Routers` and `Bulk Data Routers` execute in separate threads and processes, respectively.

Figure 13 illustrates the relationship between objects in the `Gateway` after four connections have been established. Four other connections that have not yet completed are "owned" by the `Connector`. When all `Peer` connections are completely established, the `Gateway` can route and forward messages sent to it by `Peers`.

2.10 Known Uses

The Acceptor and Connector patterns have been used in a wide range of frameworks, toolkits, and systems:

• **UNIX network superservers:** such as `inetd` [10], `listen` [6], and the `Service Configurator` daemon from the ASX framework [4]. 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 `listen` superserver always executes services in a separate process, the `inetd` superserver can be configured to allow both single-threaded and separate processes, and the `Service Configurator` supports single-threaded, multi-threaded, and multi-process execution of services.

• **CORBA ORBs:** The ORB Core layer in many implementations of CORBA [13] (such as VisiBroker and Orbix) use the Acceptor pattern to passively initialize server object implementations when clients request ORB services.

• **WWW Browsers:** The `HTML` parsing components in WWW browsers like Netscape and Internet Explorer use the asynchronous form of the Connector pattern to establish connections with servers associated with images embedded in `HTML` pages. This behavior is particularly important so that multiple HTTP connections can be initiated asynchronously to avoid blocking the browsers main event loop.

• **Ericsson EOS Call Center Management System:** this system uses the Acceptor and Connector patterns to allow application-level Call Center Manager Event Servers [14] to actively establish connections with passive Supervisors in a distributed center management system.

• **Project Spectrum:** The high-speed medical image transfer subsystem of project Spectrum [15] uses the Acceptor and Connector patterns 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.

• **ACE Framework:** Implementations of the `Reactor`, `Service Handler`, `Connector`, and `Acceptor` classes described in this paper are provided as reusable components in the ACE object-oriented network programming framework [4].

2.11 Related Patterns

The Acceptor and Connector patterns use the Template Method and Factory Method patterns [7]. The `Acceptor`'s `accept` and the `Connector`'s `connect` and `complete` functions are Template Methods that implements a generic service initialization Strategy for connecting with peers and activating a `Service Handler` when the connections is established. The use of the Template Method pattern allows subclasses to modify the specific details of creating, connecting, and activating `Service Handlers`. The Factory Method pattern is used to decouple the creation of a `Service Handler` from its subsequent use.

The Connector pattern has an intent similar to the Client-Dispatcher-Server pattern described in [16]. They both are concerned with separating active connection establishment from the subsequent service. The primary difference is that the Connector pattern addresses both synchronous and asynchronous service initialization, whereas the Client-Dispatcher-Server pattern focuses on synchronous connection establishment.

3 Concluding Remarks

This paper describes the Acceptor and Connector patterns and gives a detailed example illustrating how to use them. Implementations of the Acceptor, Connector, and Reactor patterns described in this paper are freely available via the World Wide Web at URL `www.cs.wustl.edu/~schmidt/ACE.html`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE object-oriented network programming toolkit [4] developed at Washington University, St. Louis. The ACE toolkit is currently being used on communication software at many companies including Bellcore, Siemens, DEC, Motorola, Ericsson, Kodak, and McDonnell Douglas.

Acknowledgements

Thanks to Hans Rohnert for helpful comments during the shepherding process.

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] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[3] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services," in *The* 3^{rd} *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, September 1996.

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

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

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

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

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

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

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

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

[12] D. L. Presotto and D. M. Ritchie, "Interprocess Communication in the Ninth Edition UNIX System," *UNIX Research System Papers, Tenth Edition*, vol. 2, no. 8, pp. 523–530, 1990.

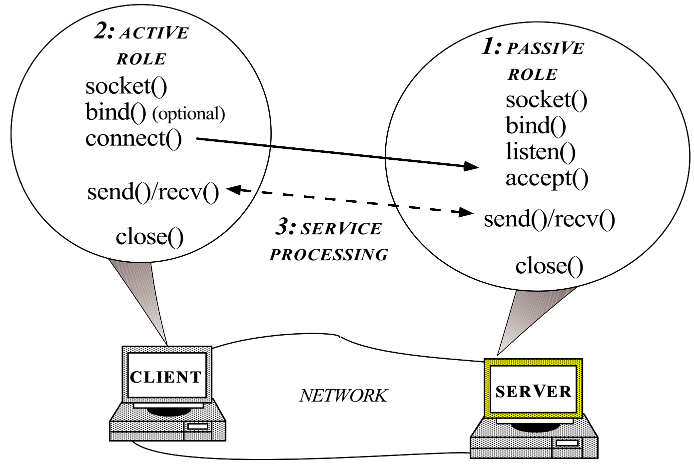

Figure 14: Active and Passive Initialization Roles

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[14] 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.

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

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

[17] J. Postel, "Transmission Control Protocol," *Network Information Center RFC 793*, pp. 1–85, Sept. 1981.

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

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

Appendix

Connection-oriented protocols (such as TCP [17]) reliably deliver data between services connected by two or more endpoints of communication. Initializing these service endpoints involves the following two roles:

- *The passive role* – which initializes a service endpoint that is listening at a particular address and waits passively for the other service endpoint(s) to connect with it;
- *The active role* – which actively initiates a connection to one or more service endpoints that are playing the passive role.

Figure 14 illustrates how these initialization roles behave and interact when a client actively connects to a passive server using the socket network programming interface [18] and the

TCP transport protocol [19]. In this figure the server plays the passive initialization role and the client plays the active initialization role.[5]

The primary goal of the Acceptor and Connector patterns is to decouple the passive and active initialization roles, respectively, from the tasks performed once the endpoints of a service are initialized. These patterns are motivated by the observation that the tasks performed on messages exchanged between endpoints of a distributed service are largely independent of the following initialization-related issues:

• **Which endpoint initiated the connection:** Connection establishment is inherently asymmetrical since the passive endpoint *waits* and the active service endpoint *initiates* the connection. Once the connection is established, however, data may be transferred between endpoints in any manner that obeys the service's communication protocol (*e.g.*, peer-to-peer, request-response, oneway streaming, etc.). Figure 14 illustrates (1) the client-side and (2) server-side connection establishment process and (3) the service processing between two connected service endpoints that exchange messages.

• **The network programming interfaces and underlying protocols used to establish the connection:** Different network programming interfaces (such as sockets [10] or TLI [6]) provide different library calls to establish connections using various underlying communication protocols (such as the Internet TCP/IP protocol or Novell's IPX/SPX). Regardless of the mechanism used to establish a connection, however, data can be transferred between endpoints using uniform message passing operations (*e.g.*, UNIX `send`/`recv` calls or Win32 `ReadFile`/`WriteFile`).

• **The creation, connection, and concurrency strategies used to initialize and execute the service:** The processing tasks performed by a service are often independent of the strategies used (1) to create an instance of a service, (2) connect the service instance to one or more peers, and (3) execute this service instance in one or more threads or processes. By explicitly decoupling these initialization strategies from the behavior of the service, the Connector and Acceptor patterns increase the potential for reusing and extending service-specific behavior in different environments.

[5]The distinction between "client" and "server" refer to *communication* roles, not necessarily to *initialization* roles. Although clients often play the active role when initiating connections with a passive server these initialization roles can be reversed, as shown in Section 2.3.