Acceptor-Connector

An Object Creational Pattern for Connecting and Initializing Communication Services

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, USA

An earlier version of this paper appeared in a chapter in the book *Pattern Languages of Program Design 3*, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

1 Intent

The Acceptor-Connector design pattern decouples connection establishment and service initialization in a distributed system from the processing performed once a service is initialized. This decoupling is achieved with three components: *acceptors*, *connectors*, and *service handlers*. A connector *actively* establishes a connection with a remote acceptor component and initializes a service handler to process data exchanged on the connection. Likewise, an acceptor *passively* waits for connection requests from remote connectors, establishing a connection upon arrival of such a request, and initializing a service handler to process data exchanged on the connection. The initialized service handlers then perform application-specific processing and communicate via the connection established by the connector and acceptor components.

2 Example

To illustrate the Acceptor-Connector pattern, consider the multi-service, application-level `Gateway` shown in Figure 1. In general, a `Gateway` decouples cooperating components in a distributed system and allows them to interact without having direct dependencies on each other. The particular `Gateway` in Figure 1 routes data between different service endpoints running on remote `Peers` used to monitor and control a satellite constellation. Each service in the `Peers` sends and receives several types of data via the `Gateway`, such as status information, bulk data, and commands. In general, `Peers` can be distributed throughout local area networks (LANs) and wide-area networks (WANs).

The `Gateway` is a router that coordinates the communication among its `Peers`. From the `Gateway`'s perspective, the `Peer` services whose data it routes differ solely in terms of their application-level communication protocols, which may use different framing formats and payload types.

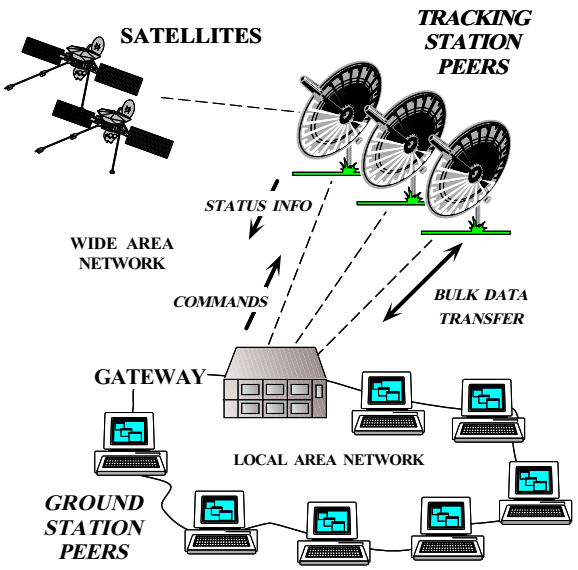

Figure 1: The Physical Architecture of a Connection-oriented Application-level Gateway

The `Gateway` transmits data between its `Peers` using the connection-oriented TCP/IP protocol [1]. In our example network configuration, each service is bound to a connection endpoint designated by an IP host address and a TCP port number. The port number uniquely identifies the type of service. Maintaining separate connections for each type of service/port increases the flexibility of routing strategies and provides more robust error handling if network connections shutdown unexpectedly.

In our distributed application, `Gateway` and `Peers` must be able to change their connection roles to support different use-cases. In particular, either may initiate a connection actively or may wait passively for connection requests. For example, in one configuration, the `Gateway` may actively initiate connections to remote `Peers` in order to route data to them. In another configuration, the `Gateway` may passively receive connection requests from `Peers` which then route data through the `Gateway` to another `Peer`. Likewise, `Peers` may be active connection initiators in one use-case and then be passive connection acceptors in another use-case.

Due to the nature of our distributed application, conventional designs that designate connection establishment and service initialization roles a priori and hard-code them into the `Gateway` and `Peer` components are too inflexible. Such a design overly couples the connection establishment, service initialization, and service processing components. This tight coupling make it hard to change connection roles independently of the communication roles.

3 Context

A client/server application in a distributed system that utilizes connection-oriented protocols to communicate between service endpoints.

4 Problem

Distributed applications often contain complex code that performs connection establishment and service initialization. In general, the processing of data exchanged between service endpoints in a distributed application is largely independent of configuration issues such as (1) which endpoint initiated the connection, *i.e.*, the *connection role* vs. the *communcation role* and (2) the connection management protocol vs. the network programming API. These issues are outlined below:

• **Connection role vs. communication role:** Connection establishment roles are inherently *asymmetrical*, *i.e.*, the passive service endpoint *waits* and the active service endpoint *initiates* the connection. Once the connection is established, however, the communication role can be *orthogonal* to the connection role. Thus, data can be transferred between service endpoints in any manner that obeys the service's communication protocol. Common communication protocols include peer-to-peer, request-response, and oneway streaming.

• **The connection management protocol vs. the network programming API:** Different network programming interfaces, such as sockets or TLI, provide different APIs to establish connections using various connection management protocols. Regardless of the protocol used to establish a connection, however, data can be transferred between endpoints using uniform message passing operations, *e.g.*, `send`/`recv` calls.

In general, the strategies for connection establishment and service initialization change much less frequently than application service implementations and communication protocols. Thus, decoupling these aspects so that they can vary independently is essential for developing and maintaining distributed applications. The following *forces* impact the solution to the problem of separating the connection and initialization protocols from the communication protocol:

- It should be easy to add new types of services, new service implementations, and new communication protocols without affecting the existing connection establishment and service initialization software. For instance, it may be necessary to extend the `Gateway` to interoperate with a directory service that runs over the IPX/SPX communication protocol, rather than TCP/IP..
- It should be possible to decouple (1) the *connection roles*, *i.e.*, which process initiates a connection vs. accepts the connection, from (2) the *communication roles*, *i.e.*, which service endpoint is the client or the server. In general, the distinction between "client" and "server" refer to communication roles, which may be orthogonal to connection roles. For instance, clients often play the active role when initiating connections with a passive server. However, these connection roles can be reversed. For example, a client that plays an active communication role may wait passively for another process to connect to it. The example in Section 2 illustrates this latter use-case.
- It should be possible to write communication software that is portable to many OS platforms in order to maximize availability and market share. Many low-level network programming APIs have semantics that are only superficially different. Therefore, it hard to write portable application using low-level APIs, such as sockets and TLI, due to syntactic incompatibilities.
- It should be possible to shield programmers from the lack of typesafety in low-level network programming APIs like sockets or TLI. For example, connection establishment code should be completely decoupled from subsequent data transport code to ensure that endpoints are used correctly. Without this strong decoupling, for instance, services may mistakenly read or write data on passive-mode transport endpoint factories that should only be used to accept connections.
- It should be possible to reduce connection latency by using OS features like asynchronous connection establishment. For instance, applications with a large number of peers may need to asynchronously establish many connections concurrently. Efficient and scalable connection establishment is particularly important for applications that run over long-latency WANs.
- It should be possible to reuse as much general-purpose connection establishment and service initialization software as possible in order to leverage prior development effort.

5 Solution

For each *service* offered by a distributed application, use the *Acceptor-Connector* pattern to decouple connection establishment and service initialization from subsequent processing performed by the two endpoints of a service once they are connected and initialized.

Introduce two factories that produce connected and initialized service handlers, which implement the application services. The first factory, called acceptor, creates and ini-

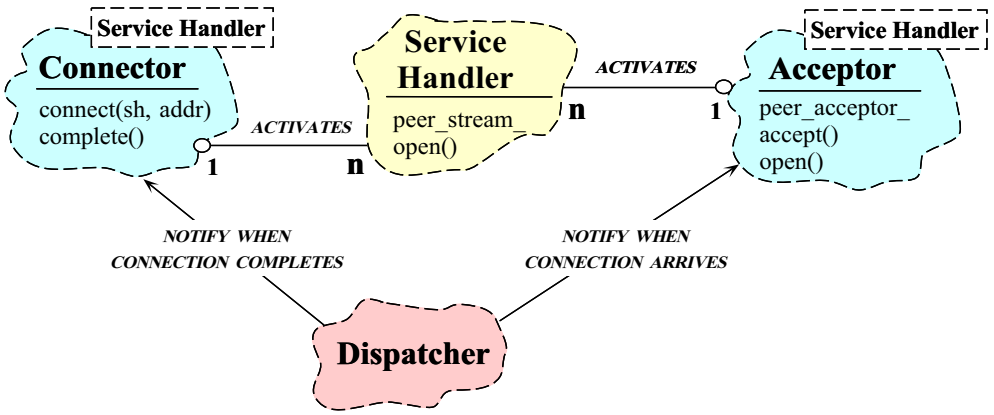

Figure 2: Structure of Participants in the Acceptor-Connector Pattern

tializes a transport endpoint that passively listens at a particular address for connection requests from remote connectors. The second factory, a connector, actively initiates a connection to a remote acceptor. Acceptor and connectors both initialize the corresponding service handlers that process the data exchanged on the connection. Once the service handlers are connected and initialized they perform the application-specific processing, and generally do not interact with the acceptor and connector any further.

6 Structure

The structure of the participants in the Acceptor-Connector pattern is illustrated by the Booch class diagram [2] in Figure 2.[1]

Service Handler: A `Service Handler` implements an application service, typically playing the client role, server role, or both roles. It provides a hook method that is called by an `Acceptor` or `Connector` to activate the application service when the connection is established. In addition, the `Service Handler` offers a data-mode transport endpoint that encapsulates an I/O handle, such as a socket. Once connected and initialized, this endpoint is used by the `Service Handler` to exchange data with its connected peer.

Acceptor: An `Acceptor` is a factory that implements the strategy for *passively* establishing a connection and initializing its associated `Service Handler`. In addition, the `Acceptor` contains a passive-mode transport endpoint factory that creates new data-mode endpoints used by `Service Handler`'s to transmit data between connected peers. The `Acceptor`'s `open` method initializes its transport endpoint factory once by binding it to a network address, such as the TCP port number the `Acceptor` is listening on.

Once initialized, the passive-mode transport endpoint factory listens for connection requests from peers. When a connection request arrives, the `Acceptor` creates a `Service Handler` and uses its transport endpoint factory to accept a new connection into the `Service Handler`.

Connector: A `Connector` is a factory that implements the strategy for *actively* establishing a connection and initializing its associated `Service Handler`. It provides a method that initiates a connection to a remote `Acceptor`. Likewise, it provides another method that finishes activating `Service Handlers` whose connections were initiated either synchronously or asynchronously. The `Connector` uses two separate methods to support asynchronous connection establishment transparently.

Both the `Acceptor` and `Connector` activate a `Service Handler` by calling its activation hook method when a connection is established. Once a `Service Handler` is completely initialized by an `Acceptor` or `Connector` factory it typically does not interact with these components any further.

Dispatcher: For the `Acceptor`, the `Dispatcher` demultiplexes connection requests received on one or more transport endpoints to the appropriate `Acceptor`. The `Dispatcher` allows multiple `Acceptors` to register with it in order to listen for connections from different peers on different ports simultaneously.

For the `Connector`, the `Dispatcher` handles the completion of connections that were initiated asynchronously. In this case, the `Dispatcher` calls back to the `Acceptor` when an asynchronous connection is established. The `Dispatcher` allows multiple `Service Handlers` to have their connections initiated and completed asynchronously by a `Connector`. Note that the `Dispatcher` is not necessary for synchronous connection establishment since the thread of control that initiates the connection also completes the service handler activation.

A `Dispatcher` is typically implemented using an event demultiplexing pattern, such those provided by the Reactor [3] or Proactor [4], which handle synchronous and asynchronous demultiplexing, respectively. Likewise, the `Dispatcher` can be implemented as a separate thread or process using the Active Object pattern [5].

[1] In this diagram 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.

7 Dynamics

The following section describes the collaborations performed by the `Acceptor` and `Connector` components in the Acceptor-Connector pattern. We examine the three canonical scenarios: for the `Acceptor`, asynchronous `Connector`, and synchronous `Connector`.

7.1 Acceptor Component Collaborations

Figure 3 illustrates the collaboration between the `Acceptor` and `Service Handler` participants. These

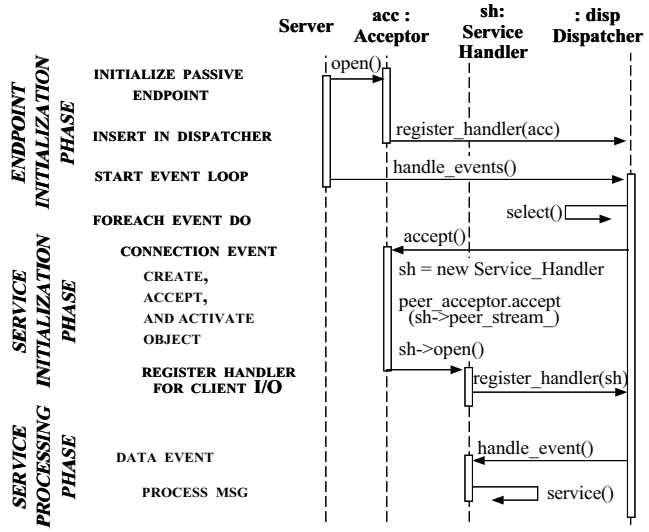

Figure 3: Collaborations Among Acceptor Participants

collaborations are divided into three phases:

1. Endpoint initialization phase: To initialize a connection passively, an application calls the `open` method of the `Acceptor`. This method creates a passive-mode transport endpoint and binds it to a network address, *e.g.* the local host's IP name and a TCP port number, and then listens for connection requests from peer `Connectors`. Next, the `open` method registers the `Acceptor` object with a `Dispatcher` so that the dispatcher can call back to the `Acceptor` when connection events arrive. Finally, the application initiates the `Dispatcher`'s event loop, which waits for connection requests to arrive from peer `Connectors`.

2. Service initialization phase: When a connection request arrives, the `Dispatcher` calls back to the `Acceptor`'s `accept` method. The `accept` method assembles the resources necessary to (1) create a new `Service Handler`, (2) use its passive-mode transport endpoint factory to accept the connection into the data-mode transport endpoint of this handler, and (3) activate the `Service Handler` by calling its `open` hook. The `open` hook of the `Service Handler` can perform service-specific initialization, such as allocating locks, spawning threads, opening log files, and/or registering the `Service Handler` with a `Dispatcher`.

3. Service processing phase: After the connection has been established passively and the `Service Handler` has been initialized, the service processing phase begins. In this phase, an application-level communication protocol, such as HTTP or IIOP, is used to exchange data between the local `Service Handler` and its connected remote `Peer` via its `peer_stream_` endpoint. When this exchange is complete the connection and `Service Handlers` can be shut down and resources released.

7.2 Connector Component Collaborations

The `Connector` component can initialize its `Service Handler` using two general schemes: *synchronous* and *asynchronous*. 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 efficient to use a different thread to connect each `Service Handler` synchronously; or
- If the services must be initialized in a fixed order and the client cannot perform useful work until connections are established.

Likewise, asynchronous service initialization is useful in opposite situations:

- If 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 being established.

The collaborations among the participants in the *synchronous* `Connector` scenario can be divided into the following three phases:

1. Connection initiation phase: To synchronously initiate a connection between a `Service Handler` and its remote `Peer`, an application calls the `Connector`'s `connect` method. This method actively establishes the connection by blocking the thread of control of the calling thread until the connection completes synchronously.

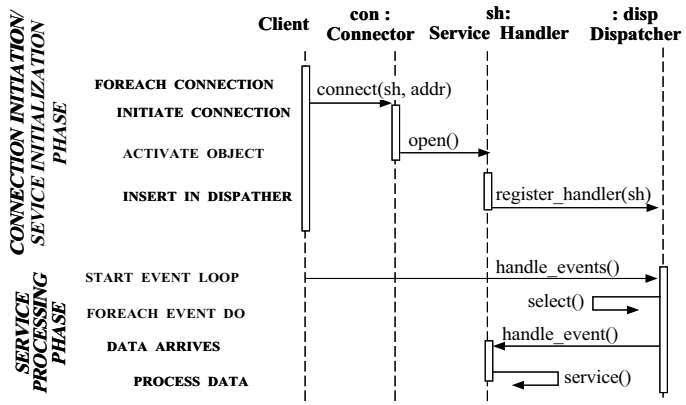

Figure 4: Collaborations Among Connector Participants for Synchronous Connections

2. Service initialization phase: After the connection completes, the `Connector`'s `connect` method calls the `complete` method to activates the `Service Handler`. The `complete` method performs the activation by invoking the `Service_Handler`'s `open` hook method, which performs service-specific initialization.

3. Service processing phase: This phase is similar to the service processing phase a `Service Handler` performs once it is created by an `Acceptor`. In particular, once a `Service Handler` is activated it performs application-specific service processing using data exchanged with the remote `Service Handler` it is connected to.

The collaboration for synchronous service initialization is shown in Figure 4. In this scheme, the `Connector` combines the connection initiation and service initialization phases into a single blocking operation. In this scenario, only one connection is established for every invocation of `connect` in each thread of control.

The collaborations among the participants in the *asynchronous* `Connector` can be divided into the following three phases:

1. Connection initiation phase: To asynchronously initiate a connection between a `Service Handler` and its remote `Peer`, an application calls the `Connector`'s `connect` method. As with the synchronous scenario, the `Connector` actively establishes the connection. However, it does not block the thread of control of the caller while the connection completes asynchronously. Instead, it registers the `Service Handler`'s transport endpoint, which we call `peer_stream_` in this example, with the `Dispatcher` and returns control to its caller.

2. Service initialization phase: After the connection completes asynchronously, the `Dispatcher` calls back the `Connector`'s `complete` method. This method activates the `Service Handler` by calling its `open` hook. This `open` hook performs service-specific initialization.

3. Service processing phase: This phase is similar to the other service processing phases described earlier. Once the `Service Handler` is activated it performs application-specific service processing using the data exchanged with the remote `Service Handler` it is connected to.

Figure 5 illustrates these three phases of collaboration using asynchronous connection establishment. In the asynchronous scenario, note how the connection initiation phase is separated temporally from the service initialization phase. This decoupling enables multiple connection initiations (via `connect`) and completions (via `complete`) to proceed in parallel within each thread of control.

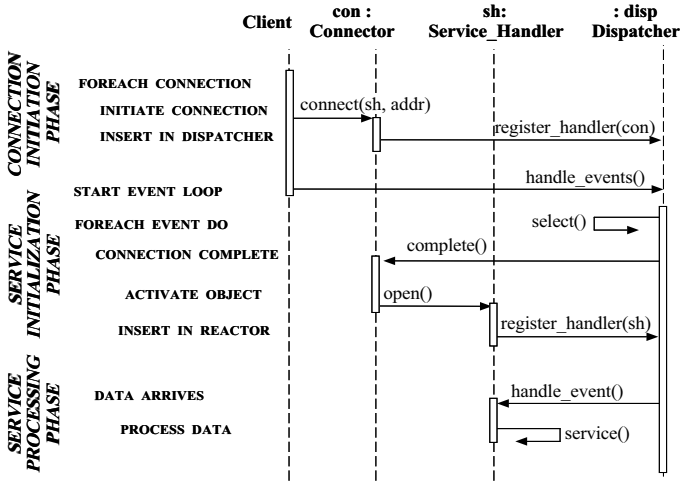

Figure 5: Collaborations Among Connector Participants for Asynchronous Connections

8 Implementation

This section explains the steps involved in building communication software applications using the Acceptor-Connector pattern. The implementation in this section is based on the reusable components and applications in the ACE OO network programming toolkit [6]. ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks across a range of OS platforms.

The participants in the Acceptor-Connector pattern are divided into the *Reactive*, *Connection*, and *Application* layers, as shown in Figure 6.[2]

The Reactive and Connection layers perform generic, application-independent strategies for dispatching events and initializing services, respectively. The Application layer instantiates these generic strategies by providing concrete

[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.

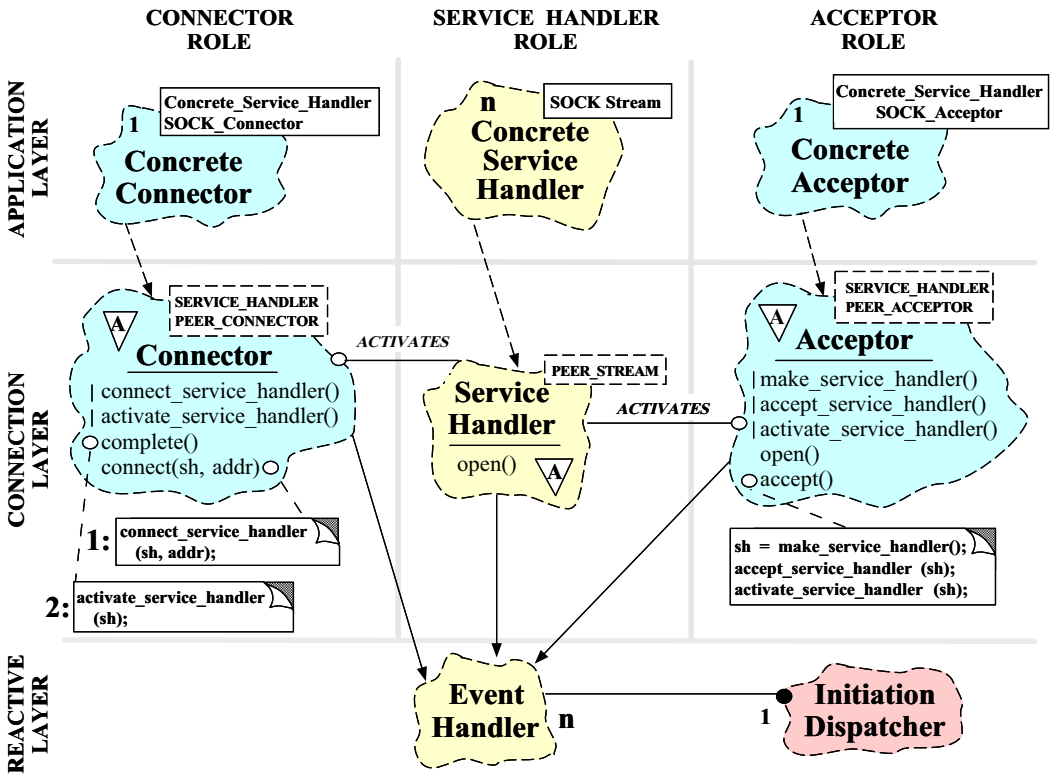

Figure 6: Layering and Partitioning of Participants in the Acceptor-Connector Pattern Implementation

classes that establish connections and perform service processing. This separation of concerns increases the reusability, portability, and extensibility in the implementation of the Acceptor-Connector pattern.

The following discussion of the Acceptor-Connector pattern implementation starts at the bottom with the Reactive layer and works upwards through the Connection layer and Application layer.

8.1 Reactive Layer

The Reactive layer handles events that occur on transport endpoints represented by I/O handles, such as socket endpoints. The two participants in this layer, the `Initiation Dispatcher` and `Event Handler`, are defined by the Reactor pattern [3]. This pattern enables efficient demultiplexing of multiple types of events from multiple sources within a single thread of control.

The two main roles in the Reactive layer are:

Event Handler: Specifies an interface consisting of hook methods [7] that abstractly represent the event processing operations that can be provided by an application. For instance, these hook methods signify events such as a new connection request, a completion of a connection request started asynchronously, or the arrival of data from a connected peer. The `Acceptor` and `Connector` components are concrete event handlers that derive from `Event Handler`.

Initiation Dispatcher: Defines an interface for registering, removing, and dispatching `Event Handlers`. The `Synchronous Event Demultiplexer`, such as `select` [8] or `WaitForMultipleObjects` [9], informs the `Initiation Dispatcher` when to call back application-specific event handlers in response to certain types of events. Common events include connection acceptance events, data input and output events, and timeout events.

Note that the `Initiation Dispatcher` is an implementation of the `Dispatcher` participant described in Section 6. In general, the Acceptor-Connector `Dispatcher` participant can be reactive, proactive, or multi-threaded. The particular `Initiation Dispatcher` in this implementation uses the reactive model to demultiplex and dispatch concrete event handlers in a single thread of control. In our example the `Initiation Dispatcher` is a Singleton [10] since we only need one instance of it for the entire process.

8.2 Connection Layer

The Connection layer

1. Creates `Service Handlers`;
2. Passively or actively connects `Service Handlers` to their remote peers; and
3. Activates `Service Handlers` once they are connected.

All behavior in this layer is completely generic. In particular, note how the classes in the implementation described below delegate to concrete IPC mechanisms and Concrete Service Handlers, which are instantiated by the Application layer described in Section 8.3.

The manner in which the Application layer delegates to the Connection layer is similar to how the Connection layer delegates to the Reactive layer. For instance, the Initiation Dispatcher in the Reactive layer processes initialization-related events, such as establishing connections asynchronously, on behalf of the Connection layer.

There are three primary roles in the Connection layer: Service Handler, Acceptor, and Connector.

Service Handler: This abstract class inherits from Event_Handler and provides a generic interface for processing services provided by clients, servers, or components that perform both roles. Applications must customize this class via inheritance to perform a particular type of service. The Service Handler interface 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;

  // Accessor method used by Acceptor and
  // Connector to obtain the underlying stream.
  PEER_STREAM &peer (void) {
    return peer_stream_;
  }

  // Return the address that we're connected to.
  PEER_STREAM::PEER_ADDR &remote_addr (void) {
    return peer_stream_.remote_addr ();
  }

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

Once the Acceptor or Connector establishes a connection, they call the open hook of a Service Handler. This pure virtual method must be defined by a Concrete Service Handler subclass, which performs service-specific initializations and subsequent processing.

Connector: This abstract class implements the generic strategy for actively establishing connections and initializing Service Handlers. The interface of the Connector is 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_CONNECTOR
```

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, either synchronously or asynchronously. The SERVICE HANDLER provides the service that processes data exchanged with its connected peer. C++ parameterized types are used to decouple (1) the connection establishment strategy from (2) the type of service handler, network programming interface, and transport layer connection protocol.

Parameterized types are an implementation decision that help improve portability. For instance, they allow the wholesale replacement of the IPC mechanisms used by the Connector. This makes the Connector's connection establishment code portable across platforms that contain different network programming interfaces, *e.g.*, 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 [11]. Another motivation for using parameterized types is to im-

prove run-time efficiency since template instantiation occurs at compile-time.

An even more dynamic type of decoupling could be achieved via inheritance and polymorphism by using the Factory Method and Strategy patterns described in [10]. For instance, a `Connector` could store a pointer to a `PEER CONNECTOR` base class. The `connect` method of this `PEER CONNECTOR` could be dynamically bound at run-time in accordance with the subclass of `PEER CONNECTOR` returned from a Factory. In general, the tradeoff between parameterized types and dynamic binding is that parameterized types can incur additional compile/link-time overhead, whereas dynamic binding can incur additional run-time overhead.

The `connect` method is the entry point an application uses to initiate a connection via a `Connector`. It's implementation is shown below.[3]

```
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 uses the Bridge pattern [10] to allow `Concrete Connectors` to transparently modify the connection strategy, without changing the component interface. Therefore, the `connect` method delegates to the `Connector`'s connection strategy, `connect_service_handler`, which initiates a connection as shown below:

```
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 connection doesn't complete immediately
      // and we are using non-blocking semantics
      // then register this object  with the
      // Initiation_Dispatcher Singleton so it will
      // callback when the connection is complete.
      Initiation_Dispatcher::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);
}
```

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

If the value of the `Connect_Mode` parameter is `SYNC` the `SERVICE HANDLER` will be activated once the connection completes synchronously, as illustrated in Figure 7. This figure is similar to Figure 4, but provides additional implementation details, such as the use of the `get_handle` and `handle_event` hook methods.

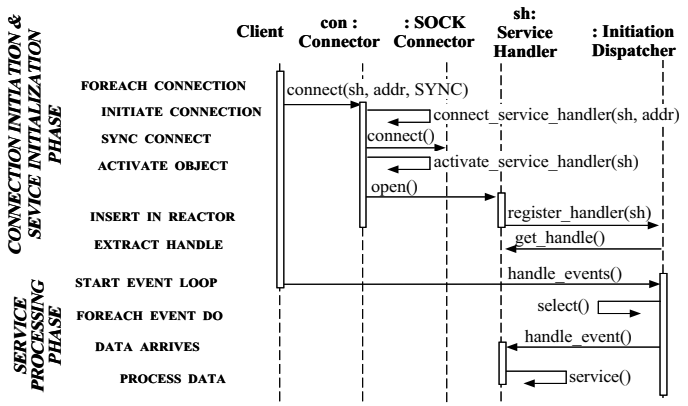

Figure 7: Collaborations Among the Connector Participants for Synchronous Connections

To connect with multiple `Peers` efficiently, the `Connector` also may need 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 8. This figure is similar to Figure 5, but it also provides additional details that correspond to the current implementation.

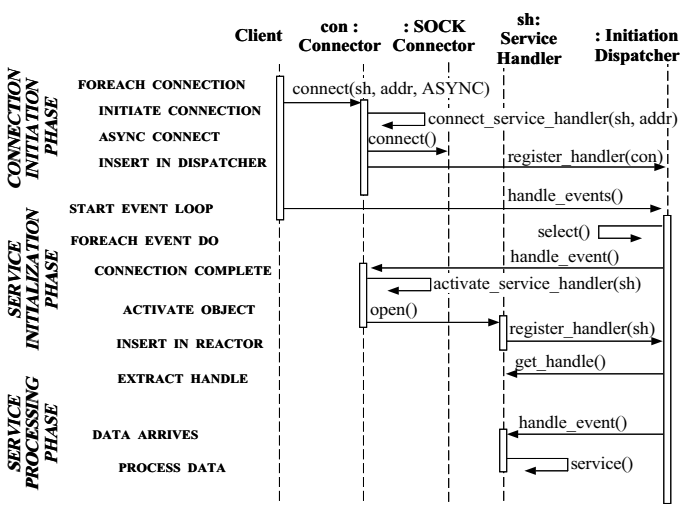

Figure 8: Collaborations Among the Connector Participants for Asynchronous Connections

Once instantiated, the `PEER CONNECTOR` class provides the concrete IPC mechanism to initiate connections

synchronously or asynchronously. The implementation of the Connector pattern shown here uses asynchronous connection mechanisms provided by the OS and communication protocol stack. For instance, on UNIX or Win32 the Connector can set sockets into non-blocking mode and use an event demultiplexer like select or WaitForMultipleObjects to determine when the connection completes.

To handle asynchronous connections that are still pending completion, the Connector maintains a map of Service Handlers. Since the Connector inherits from Event Handler, the Initiation Dispatcher can automatically call back to the Connector's handle_event method when a connection completes.

The handle_event method is an Adapter [10] that transforms the Initiation Dispatcher's event handling interface to a call to the Connector pattern's complete method.

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 Initiation_Dispatcher's event
  // handling API to the Connector's API.
  complete (handle);
}
```

The complete method activates the SERVICE HANDLER whose non-blocking connection just completed successfully, as follows:

```
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 Initiation_Dispatcher.
  Initiation_Dispatcher::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 and transfers the I/O HANDLE to the SERVICE HANDLER. Finally, it initializes the SERVICE HANDLER by calling activate_service_handler. This method delegates to the concurrency strategy designated by the SERVICE HANDLER's open hook, as follows:

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

The Service Handler's open hook is called when a connection is established successfully. Note that it is called regardless of whether (1) connections are initiated synchronously or asynchronously or (2) they are connected actively or passively. This uniformity makes it possible to write Service Handlers whose processing can be completely decoupled from how they are connected and initialized.

Acceptor: This abstract class implements the generic strategy for passively establishing connections and initializing Service Handlers. The interface of the Acceptor is 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 transport endpoint factory
  //  and register with Initiation_Dispatcher Singleton.
  virtual int open
    (const PEER_ACCEPTOR::PEER_ADDR &local_addr);

  // Factory Method 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,
  // which is used by Initiation_Dispatcher 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 remote peer. Note that the SERVICE HANDLER is a concrete service handler that is provided by the application layer.

Parameterized types decouple the Acceptor's 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 more efficiently.

The implementation of the Acceptor's methods is presented below. Applications initiatize an Acceptor by calling its open method, 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 Initiation_Dispatcher, which
  // ``double-dispatches'' without get_handle()
  // method to extract the HANDLE.
  Initiation_Dispatcher::instance
    ()->register_handler (this, READ_MASK);
}
```

The open method is passed a local_addr. This parameter contains a network address, *e.g.*, the local host's IP name and TCP port number, used to listen for connections. It forwards this address to the passive connection acceptance mechanism defined by the PEER ACCEPTOR. This mechanism initializes the transport endpoint factory, which advertises its address to clients who are interested in connecting with the Acceptor.

The behavior of the transport endpoint factory is determined by the type of PEER ACCEPTOR instantiated by a user. For instance, it can be a C++ wrapper [12] for sockets [13], TLI [14], STREAM pipes [15], Win32 Named Pipes, etc.

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

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

The Initiation Dispatcher stores this HANDLE internally in a table. A Synchronous Event Demultiplexer, such as select, is then used to detect and demultiplex incoming connection requests from clients. Since the Acceptor class inherits from Event Handler, the Initiation Dispatcher 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 Initiation Dispatcher'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 Initiation_Dispatcher's event handling
  // API to the Acceptor's API.
  accept ();
}
```

As shown below, the accept method is a Template Method [10] that implements the Acceptor-Connector 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, which are instantiated via parameterized types and can be customized by subclasses of the Acceptor. In particular, since the accept is a Template Method, subclasses can extend any or all of the Acceptor's connection establishment and initialization strategies. This flexibility makes it possible to write Service Handlers whose behavior is decoupled from the manner in which they are passively connected and initialized.

The make_service_handler factory method defines the default strategy an Acceptor uses to create SERVICE HANDLERs, as follows:

```
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 [10] 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->peer ());
}
```

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 choose its own concurrency strategy. For instance, if the SERVICE HANDLER inherits from Event Handler it can register with the Initiation Dispatcher. This allows the Initiation Dispatcher to dispatch the SERVICE HANDLER's handle_event method when events occur on its PEER STREAM endpoint of communication. Concrete Acceptors can override this strategy to do more sophisticated concurrency activations. For instance, a subclass could make the SERVICE HANDLER an Active Object [5] 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 Initiation Dispatcher calls the Acceptor's handle_close method, which can release any dynamically acquired resources. In this case, the handle_close method simply forwards the close request to the PEER ACCEPTOR's transport endpoint factory, as follows:

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

8.3 Application Layer

The Application Layer supplies concrete interprocess communication (IPC) mechanisms and a concrete Service Handler. 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 9 are part of the ACE C++ socket wrapper library [11]. These wrappers encapsulate 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 defines the concrete application 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.

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.

Concrete Service Handlers can also define a service's concurrency strategy. For example, a Service Handler may inherit from the Event Handler and employ the Reactor [3] pattern to process data from peers in a single-thread of control. Conversely, a Service Handler might use the Active Object pattern [5] to process incoming data in a different thread of control than the one used by the Acceptor to connect it. Below, we implement Concrete Service Handlers for our Gateway example, illustrates how several different concurrency strategies can be configured flexibly without affecting the structure or behavior of the Acceptor-Connector pattern.

In the sample code in Section 9, SOCK Connector and SOCK Acceptor are the IPC mechanisms used to establish connections actively and passively, respectively. Likewise, a SOCK Stream is used as the data transport delivery mechanism. 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. Likewise, it is easy to vary the data transfer mechanism by parameterizing the Concrete Service Handler with a different PEER STREAM, such as an SVR4 UNIX TLI Stream or a Win32 Named Pipe Stream.

Section 9 illustrates how to instantiate a Concrete Service Handler, Concrete Connector, and Concrete Acceptor that implement the Peers and Gateway described in Section 2. This particular example of the Application layer customizes the generic initialization strategies provided by the Connector and Acceptor components in the Connection layer.

9 Example Resolved

The code below illustrates how the Peers and Gateway described in Section 2 can use the Acceptor-Connector pat-

terns to simplify connection establishment and service initialization. Section 9.1 illustrates how the Peers play a passive role and Section 9.2 illustrates how the Gateway plays an active role in establishing connections with the passive Peers.

9.1 Concrete Components for Peers

Figure 9 illustrates how the Concrete Acceptor and Concrete Service Handler components are structured in a Peer. The Acceptor components in this figure

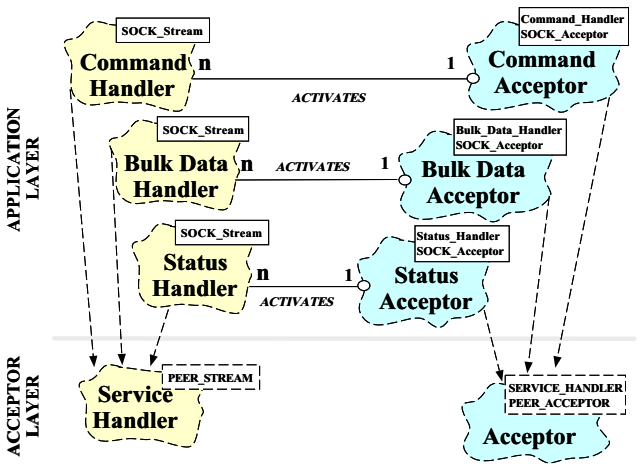

Figure 9: Structure of Acceptor Participants for Peers

complement the Connector components in Figure 11.

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

To illustrate the flexibility of the Acceptor-Connector 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 Initiation Dispatcher 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 handles. For instance, the Status Handler class processes status data sent to and received from a Gateway:

```
class Status_Handler : public PEER_HANDLER
{
public:
  // Performs handler activation.
  virtual int open (void) {
    // Make handler run in separate thread (note
    // that  Thread::spawn requires a pointer to
    // a static method as the 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 PEER HANDLER also can be subclassed to produce concrete service handlers that process bulk data and commands. For instance, the Bulk Data Handler 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 Command Handler 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
    // Initiation_Dispatcher singleton.
    Initiation_Dispatcher::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
    // Initiation_Dispatcher.
    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 factory instances 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_acceptor;

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

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

Note how the use of templates and dynamic binding permits specific details to change flexibly. In particular, no Acceptor component changes when the concurrency strategy is modified throughout this section. The reason for this flexibility is that the concurrency strategies have been factored out into the Service Handlers, rather than coupled with the Acceptors.

The Peer main function: The main program initializes the concrete Acceptor factories by calling their open hooks with the TCP ports for each service. Each Acceptor factory automatically registers itself with an instance of the Initiation Dispatcher in its open method, as shown in Section 8.2.

```
// Main program for the Peer.

int main (void)
{
  // Initialize acceptors with their
  // well-known ports.
  s_acceptor.open (INET_Addr (STATUS_PORT));
  bd_acceptor.open (INET_Addr (BULK_DATA_PORT));
  c_acceptor.open (INET_Addr (COMMAND_PORT));

  // Event loop that handles connection request
  // events and processes data from the Gateway.

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

Once the Acceptors are initialized, the main program enters an event loop that uses the Initiation Dispatcher to detect connection requests from the Gateway. When connections arrive, the Initiation Dispatcher 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 10 illustrates the relationship between Concrete Acceptor components in the Peer after four connections have been established with the Gateway shown in Figure 12 and four Service Handler have been created and activated. While the Concrete Service Handlers exchange data with the Gateway, the three Acceptors continue to listen for new connections in the main thread.

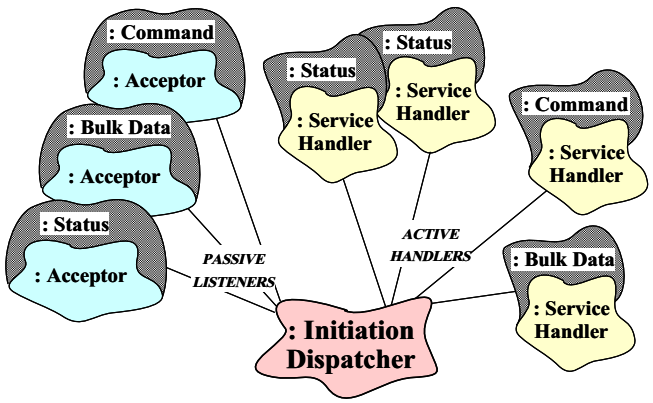

Figure 10: Object Diagram for Acceptor Components in the Peer

9.2 Concrete Components for the Gateway

Figure 11 illustrates how the Concrete Connector and Concrete Service Handler components are structured in a hypothetical configuration of the Gateway. The Connector components in this figure complement the Acceptor components in Figure 9.

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 Acceptor-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 Iniitation Dispatcher 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

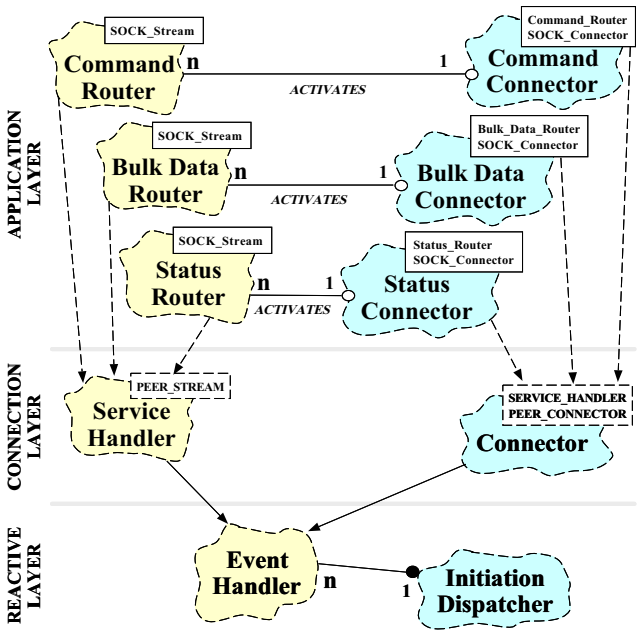

Figure 11: Structure of Connector Participants for the `Gateway`

`Connector`, which is generic and thus highly flexible and reusable.

We 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 to or from `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 `PEER ROUTER` can be subclassed to produce concrete service handlers for routing bulk data and commands. For instance, the `Bulk Data Router` routes bulk data to or from `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 to or from `Peers`:

```
class Command_Router : public PEER_ROUTER
{
public:
  // Activates router in same thread as Connector.
  virtual int open (void) {
    Initiation_Dispatcher::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 Initiation_Dispatcher.
    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;
```

Unlike the `Concrete Acceptor` components, we only require a single `Concrete Connector`. The reason for this is that each `Concrete Acceptor` is used as a factory to create a specific type of `Concrete Service Handler`, such as a `Bulk Data Handler` or a `Command Handler`. Therefore, the complete type must be known *a priori*, which necessitates multiple `Concrete Acceptor` types. In contrast, the `Concrete Service Handlers` passed to the `Connector`'s `connect` method are initialized externally. Therefore, they can be treated uniformly as `PEER ROUTERS`.

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 or naming service. 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 by passing the `ASYNC` flag to the `connect` method.

```
// Main program for the Gateway.

// Obtain an STL vector of Status_Routers,
// Bulk_Data_Routers, and Command_Routers
// from a config file.

void get_peer_addrs (vector<PEER_ROUTERS> &peers);

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

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

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

  // Iterate through all the Routers and
  // initiate connections asynchronously.

  for (vector<PEER_ROUTER>::iterator i = peers.begin ()

      i != peers.end ();
      i++) {
    PEER_ROUTER &peer = *i;
    peer_connector.connect (peer,
                            peer.remote_addr (),
                            PEER_CONNECTOR::ASYNC);
  }
  // Loop forever handling connection completion
  // events and routing data from Peers.

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

All connections are invoked asynchronously. They complete concurrently via the `Connector`'s `complete` method, which is called back within the event loop of the `Initiation Dispatcher`. This event loop also demultiplexes and dispatches routing events for `Command Router` objects, which run in the `Initiation Dispatcher`'s thread of control. The `Status Routers` and `Bulk Data Routers` execute in separate threads and processes, respectively.

Figure 12 illustrates the relationship between components in the `Gateway` after four connections have been established with the `Peer` shown in Figure 10 and four `Concrete Service Handlers` have been created and activated. This diagram illustrates four connections to another `Peer` that have not yet completed are "owned" by the `Connector`. When all `Peer` connections are completely established, the `Gateway` will route and forward messages sent to it by `Peers`.

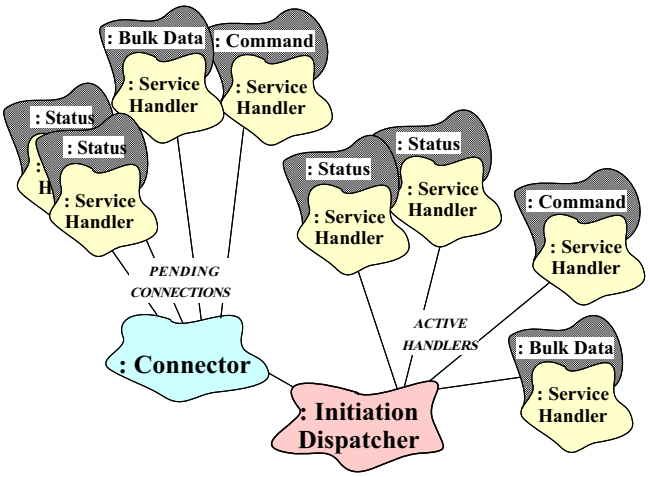

Figure 12: Object Diagram for Connector Components in the Gateway

10 Known Uses

The Acceptor-Connector pattern has been used in a wide range of frameworks, toolkits, and systems:

UNIX network superservers: such as `inetd` [13], `listen` [14], and the `Service Configurator` daemon from the `ASX` framework [6]. 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 process decouples the functionality of the `inetd` superserver into two separate parts: one for establishing connections and another for receiving and processing requests from peers. When a service request arrives on a monitored port, the Acceptor process accepts the request and dispatches an appropriate pre-registered handler to perform the service.

CORBA ORBs: The ORB Core layer in many implementations of CORBA [16] use the Acceptor-Connector to passively initialize server object implementations when clients request ORB services. [17] describes how the Acceptor-Connector pattern is used to implement The ACE ORB (TAO) [18], which is a real-time implementation of CORBA.

WWW Browsers: The `HTML` parsing components in WWW browsers like Netscape and Internet Explorer use the asynchronous version of the `connector` component to establish connections with servers associated with images embedded in `HTML` pages. This 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-Connector pattern to allow application-level Call Center Manager Event Servers [19] 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 [20] uses the Acceptor-Connector pattern to passively establish connections and initialize application services for storing large medical images. Once connections are established, applications then send and receive multi-megabyte medical images to and from these image stores.

ACE Framework: Implementations of the `Service Handler`, `Connector`, and `Acceptor` classes described in this paper are provided as reusable components in the ACE object-oriented network programming framework [6].

11 Consequences

11.1 Benefits

The Acceptor-Connector pattern provides the following benefits:

Enhances the reusability, portability, and extensibility of connection-oriented software by decoupling mechanisms for initializing services from subsequent service processing. For instance, the application-independent mechanisms in the `Acceptor` and `Connector` are reusable components that know how to (1) establish connections passively and actively, respectively and (2) initialize the associated `Service Handler` once the connection is established. 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 small number of components 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 transport endpoint factory `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 [11].

Efficiently utilize the inherent parallelism in the network and hosts: By using the asynchronous mechanisms shown in Figure 8, 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 7. 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.

11.2 Drawbacks

The Acceptor-Connector pattern has the following drawbacks:

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

12 See Also

The Acceptor-Connector pattern use the Template Method and Factory Method patterns [10]. The `Acceptor`'s `accept` and the `Connector`'s `connect` and `complete` functions are Template Methods that implements a generic service strategy for connecting to remote peers and initializing a `Service Handler` when the connection is established. The use of the Template Method pattern allows subclasses to modify the specific details of creating, connecting, and activating `Concrete Service Handlers`. The Factory Method pattern is used to decouple the creation of a `Service Handler` from its subsequent use.

The Acceptor-Connector pattern has an intent similar to the Client-Dispatcher-Server pattern described in [21]. They both are concerned with separating active connection establishment from the subsequent service. The primary difference is that the Acceptor-Connector pattern addresses passive and active service initialization for both synchronous and asynchronous connections, whereas the Client-Dispatcher-Server pattern focuses on synchronous connection establishment.

Acknowledgements

Thanks to Frank Buschmann and Hans Rohnert for their helpful comments on this paper.

References

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

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

[3] 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.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[4] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The 4^{th} Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34)*, September 1997.

[5] 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.

[6] 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.

[7] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.

[8] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

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

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

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

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

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

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

[15] 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.

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

[17] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *Submitted to the IEEE Communications Magazine*, 1998.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[19] 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.

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

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