Connector

A Design Pattern for Actively Initializing Network Services

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

This paper appeared in the January 1996 issue of the C++ Report magazine.

1 Introduction

This article is part of a continuing series that describes object-oriented techniques for developing reusable, extensible, and efficient communication software. The topic of this article is the *Connector pattern*. This design pattern enables the tasks performed by network services to evolve independently of the mechanisms that *actively* initialize the services. The Connector pattern is a companion to the *Acceptor pattern* [1], which enables network services to evolve independently of the mechanisms that *passively* establish connections used by the services.

The Connector and Acceptor patterns are commonly used in conjunction with connection-oriented protocols (such as TCP or SPX). These protocols reliably deliver data between two communication endpoints. Establishing connections between two endpoints involves both a *passive role* and an *active role*. The passive role initializes an endpoint of communication at a particular address (such as an Internet IP address and port number) and waits passively for other endpoints to connect with it. The active role initiates a connection to the address of an endpoint playing the passive role.

The intent of the Connector and Acceptor patterns is to decouple the active and passive connection roles, respectively, from the network services performed once connections are established. Common connection-oriented network services include remote login, file transfer, and access to World-Wide Web resources. This article describes how separating the connection-related processing from the service processing yields more reusable, extensible, and efficient communication software.

This article is organized as follows: Section 2 motivates the Connector pattern by illustrating how it can be used to actively establish connections with a large number of peers in a connection-oriented, multi-service, application-level `Gateway`; Section 3 describes the Connector pattern in detail and illustrates one way to implement it in C++; and Section 4 presents concluding remarks.

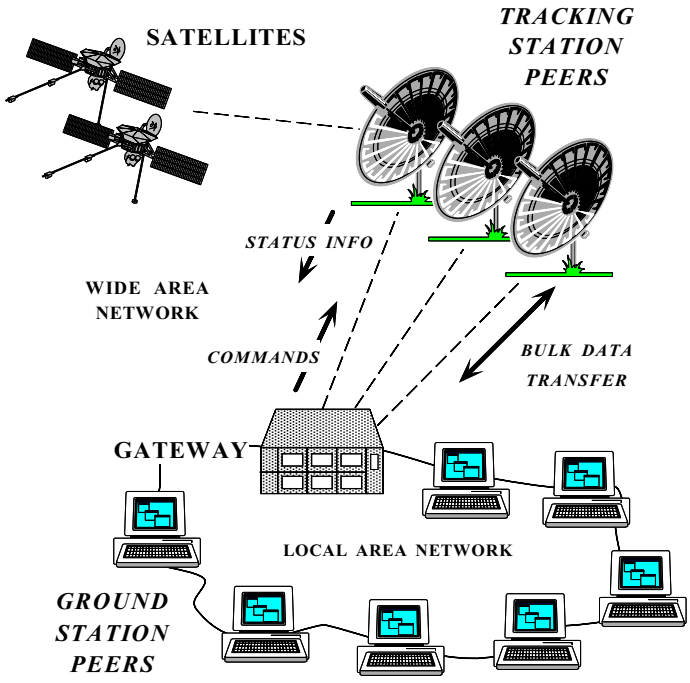

Figure 1: A Connection-oriented, Multi-service Application-level Gateway

2 Motivation

To illustrate the Connector pattern, consider the multi-service, application-level `Gateway` shown in Figure 1. This `Gateway` routes several types of data (such as status information, bulk data, and commands) between network services running on `Peers` that monitor and control a satellite constellation.[1] These `Peers` are located throughout local area networks (LANs) and wide-area networks (WANs).

The `Gateway` is connected to the `Peers` via reliable, connection-oriented interprocess communication (IPC) protocols such as TCP. Using a connection-oriented protocol simplifies application error handling and enhances performance over long-delay WANs. Each communication service

[1] In design patterns terminology, the `Gateway` is a Mediator [2] that coordinates interactions between its connected `Peers`.

in the `Peers` sends and receives status information, bulk data, and commands to and from the `Gateway` using separate TCP connections. The different services are connected to unique port numbers. 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 policies and more robust error handling when connections fail.

In this system, the `Gateway` is responsible for initiating connections to the `Peers`. Thus, the `Gateway` plays the *active* connection role and the `Peers` play the *passive* role. In a large configuration, the `Gateway` must connect to dozens or hundreds of `Peers`. Once the connections are established, the `Gateway` routes data from `Peer` to `Peer` for each type of service it supports.

To decouple the various types of routing services from the mechanisms that actively establish connections, the `Gateway` uses the *Connector pattern*. This pattern resolves the following forces for network clients that explicitly use connection-oriented communication protocols:

- *How to reuse active connection establishment code for each new service* – The Connector pattern permits key characteristics of services (such as the concurrency strategy or the data format) to evolve independently and transparently from the mechanisms used to establish the connections. Since service characteristics change more frequently than connection establishment mechanisms this separation of concerns helps reduce software coupling and increases code reuse.
- *How to make the connection establishment code portable across platforms that contain different network programming interfaces* – This is particularly important for asynchronous connection establishment, which is hard to program portably and correctly using lower-level network programming interfaces (such as sockets and TLI).
- *How to actively establish connections with large number of peers efficiently* – The Connector pattern can employ asynchrony to initiate and complete multiple connections in non-blocking mode. By using asynchrony, the Connector pattern enables applications to actively establish connections with a large number of peers efficiently over long-delay WANs.
- *How to enable flexible service concurrency policies* – Once a connection is established, peer applications use the connection to exchange data to perform some type of service (*e.g.*, remote login, WWW HTML document transfer, etc.). A service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connection was established.

The following section describes the Connector pattern in detail, using a modified form of the design pattern description format presented in [2].

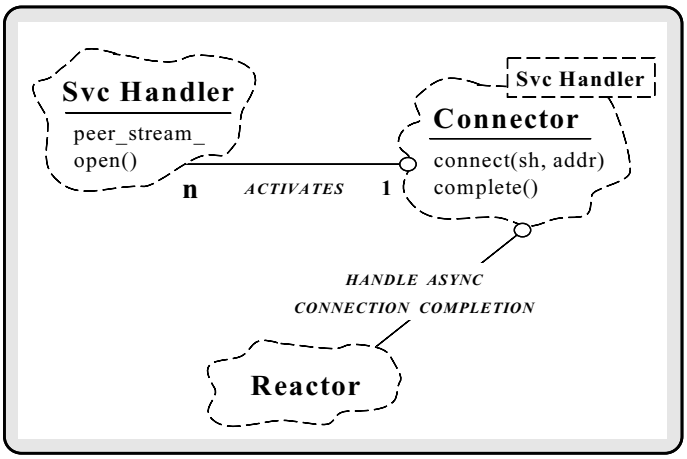

Figure 2: Structure of Participants in the Connector Pattern

3 The Connector Pattern

3.1 Intent

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

3.2 Applicability

Use the Connector pattern when connection-oriented applications have either or both of the following characteristics:

- The behavior of a network service does not depend on the steps required to actively initialize a service;
- An application must establish a large number of connections with peers connected over long-delay networks (such as satellite WANs).

3.3 Structure and Participants

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

- **Connector**
  - Connects and activates a `Svc Handler`. The `Connector`'s `connect` method implements the strategy for actively connecting the `Svc Handler` with its remote peer. The `complete` method is used to activate `Svc Handlers` whose connections were initiated and completed asynchronously.
- **Svc Handler**

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

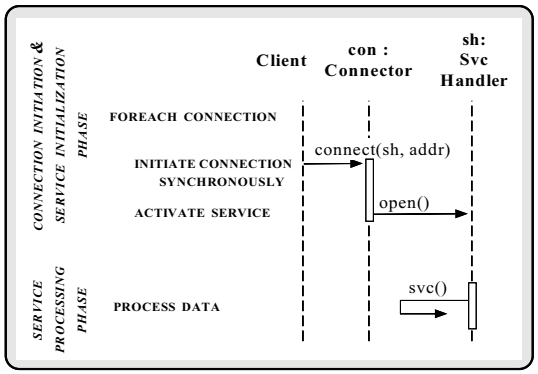

Figure 3: Collaborations Among Participants for Synchronous Connections

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

- **Reactor**

  - Handles the completion of connections that were initialized asynchronously. The `Reactor` allows multiple `Svc Handlers` to have their connections initiated and completed asynchronously by a `Connector` configured within a single thread of control.

3.4 Collaborations

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

1. *Connection initiation phase* – which actively connects one or more `Svc Handlers` with their peers. Connections can either be initiated synchronously or asynchronously. The `Connector` determines the *strategy* for actively establishing connections.

2. *Service initialization phase* – which activates a `Svc Handler` by calling its `open` method when the connection associated with it completes successfully. The `open` method of the `Svc Handler` performs service-specific initialization.

   Figures 3 and 4 illustrate the collaborations between components for synchronous and asynchronous connection initiation, respectively. The synchronous form combines connection initiation and service initialization, whereas the asynchronous form splits them into two phases. Note, however, that the steps in the service initialization and service processing phases are independent of whether the connection was initiated synchronously or asynchronously.

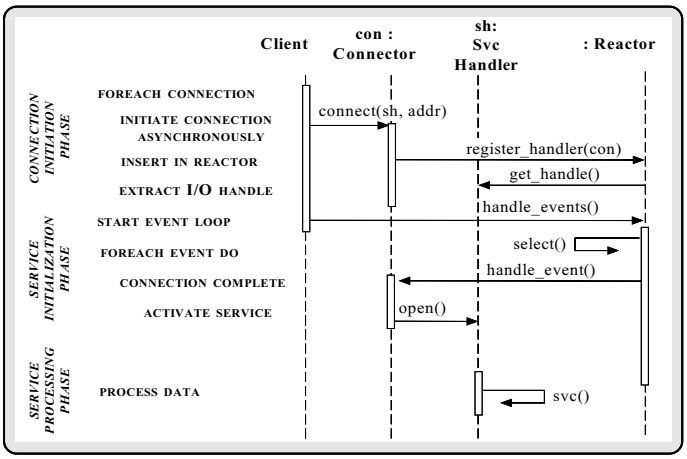

Figure 4: Collaborations Among Participants for Asynchronous Connections

3. *Service processing phase* – Once the connection has been established actively and the service has been initialized, the application enters into a *service processing phase*. This phase performs application-specific tasks that process the data exchanged between the `Svc Handler` and its connected peer(s).

3.5 Consequences

The Connector pattern provides the following benefits:

- *Enhances the reusability, portability, and extensibility of connection-oriented software* – The application-independent mechanisms for actively establishing connections are decoupled from application-specific services. Thus, the application-independent mechanisms in the `Connector` are reusable components that know how to establish a connection actively and activate its associated `Svc Handler`. In contrast, the `Svc Handler` knows how to perform application-specific service processing.

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

- *Efficiently utilize the inherent parallelism in the network and hosts* – 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 3. However, the round trip delay for a 3-way TCP connection handshake over a long-delay WAN, such as a geosynchronous satellite, 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. In contrast, by using the asynchronous mechanisms shown in Figure 4, the Connector pattern can actively establish connections with a large number of peers efficiently over long-delay WANs.

The Connector pattern has the following drawbacks:

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

3.6 Implementation

This section describes how to implement the Connector pattern in C++. The implementation described below is based on the ACE OO network programming toolkit [4]. In addition to illustrating how to implement the Connector pattern, this section shows how the pattern interacts with other common communication software patterns provided by ACE.

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

There is a striking similarity between the structure of the Connector class diagram and the Acceptor class diagram shown in [1]. In particular, the `Reactive Layer` is identical in both and the roles of the `Svc Handler` and `Concrete Svc Handler` are also very similar. Moreover, the `Connector` and `Concrete Connector` play roles equivalent to the `Acceptor` and `Concrete`

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

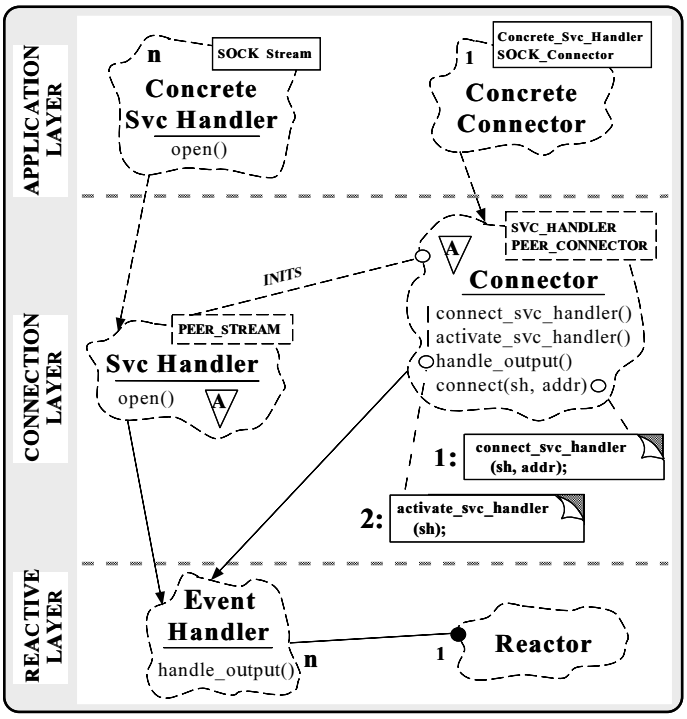

Figure 5: Layering of Participants in the Connector Pattern

`Acceptor` classes. In the Connector pattern, however, these two classes play an *active* role in establishing a connection, rather than a *passive* role.

3.6.1 Reactive Layer

The Reactive layer is responsible for handling events that occur on endpoints of communication represented by I/O handles (also known as "descriptors"). The two participants at this layer, the `Reactor` and `Event Handler`, are reused from the Reactor pattern [5]. This pattern encapsulates OS event demultiplexing system calls (such as `select`, `poll` [6], and `WaitForMultipleObjects` [7]) with an extensible and portable callback-driven object-oriented interface. The Reactor pattern enables efficient demultiplexing of multiple types of events from multiple sources within a single thread of control. An implementation of the Reactor pattern is shown in [8] and the two omain roles in the Reactive layer are describe below.

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

• **Event Handler:** This class specifies an interface that the `Reactor` uses to dispatch callback methods defined by objects that are pre-registered to handle events. These events

signify conditions such as the completion of an asynchronous connection or the arrival of data from a connected peer.

3.6.2 Connection Layer

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

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

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

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

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

The `open` method of a `Svc Handler` is called by the `Connector` factory after a connection is established. The behavior of this pure virtual method must be defined by a subclass, which performs service-specific initializations. A subclass of `Svc Handler` is also responsible for determining the service's concurrency strategy. For example, a `Svc Handler` may employ the Reactor [5] pattern to process data from peers in a single-thread of control. To enable this, `Svc Handler` inherits from the Reactor pattern's `Event Handler`, which the `Reactor` to dispatch its `handle_event` method when events occur on the `PEER STREAM` endpoint of communication. Conversely, a `Svc Handler` might use the Active Object pattern [9] to process incoming data in a different thread of control than the one the `Connector` object used to connect it. Section 3.7 illustrates several different concurrency policies.

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

```
template <class SVC_HANDLER, // Type of service
          class PEER_CONNECTOR> // Active Conn. Mech.
class Connector : public Event_Handler
{
public:
  enum Connect_Mode {
    SYNC, // Initiate connection synchronously.
    ASNYC // Initiate connection asynchronously.
  };

  // Initialization method stores Reactor *.
  Connector (Reactor *r): reactor_ (r) {}

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

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

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

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

  // Activate a SVC_HANDLER whose non-blocking
  // connection has completed successfully.
  virtual int handle_event (HANDLE sd);

protected:
  // Event demultiplexor.
  Reactor *reactor_;

  // IPC mech. that establishes connections actively.
  PEER_CONNECTOR connector_;

  // Collection that maps HANDLEs to SVC_HANDLER *s.
  Map_Manager<HANDLE, SVC_HANDLER *> handler_map_;
};

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

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

The use of parameterized types 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 [2]. 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 methods is presented below. To save space, most of the error handling has been omitted.

The main entry point for a `Connector` is `connect`:

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

This method delegates to the `Connector`'s connection strategy, `connect_svc_handler`, which initiates a connection:

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

  if (connector_.connect (*svc_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 so
      // that it will callback when the
      // connection is complete.
      reactor_->register_handler (this, WRITE_MASK);

      // Store the SVC_HANDLER in the map of
      // pending connections.
      handler_map_.bind
        (connector_.get_handle (), svc_handler);
  }
  else
    // Activate if we connect synchronously.
    activate_svc_handler (svc_handler);
}
```

If the value of the `Connect_Mode` parameter is `SYNC` the `SVC HANDLER` will be activated after the connection completes synchronously, as illustrated in Figure 6.

To connect with multiple `Peers` efficiently, however, 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 7.

The concrete `PEER CONNECTOR` class provides the low-level 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 mode).

The `Connector` maintains a map of `Svc Handlers` whose asynchronous connections are pending completion. Once an asynchronous connection completes successfully the

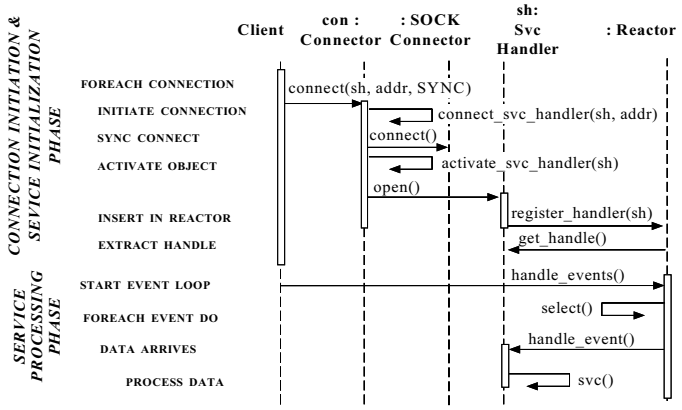

Figure 6: Collaborations Among Participants for Synchronous Connections

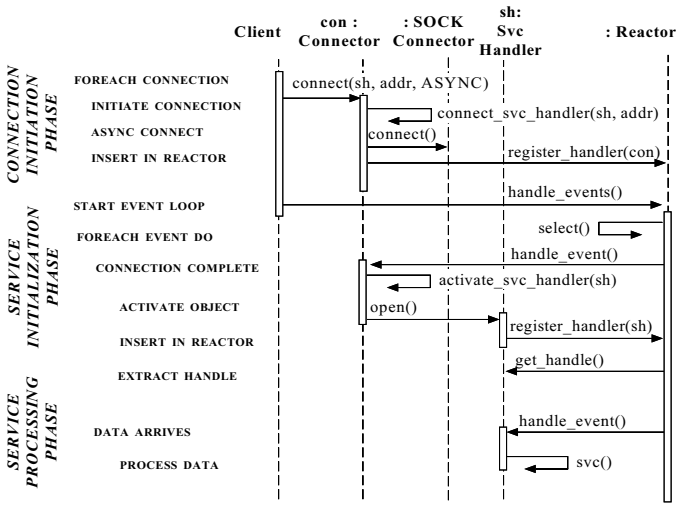

Figure 7: Collaborations Among Participants for Asynchronous Connections

Reactor calls back to the Connector's handle_event method:

```
// Activate a SVC_HANDLER whose non-blocking
// connection has completed successfully.

template <class SH, class PC> int
Connector<SH, PC>::handle_event (HANDLE handle)
{
  SVC_HANDLE *svc_handler = 0;

  // Locate the SVC_HANDLER corresponding
  // to the HANDLE.
  handler_map_.find (handle, svc_handler);

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

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

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

  // Connection is complete, so activate handler.
  activate_svc_handler (svc_handler);
}
```

The handle_events method finds and removes the connected svc handler from its internal map, transfers the I/O HANDLE to the svc handler, and initializes it by calling activate_svc_handler. This method delegates the concurrency strategy designated by the SVC HANDLER::open method:

```
template <class SH, class PC> int
Connector<SH, PC>::activate_svc_handler
  (SVC_HANDLER *svc_handler)
{
  svc_handler->open ();
}
```

Note that active_svc_handler 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 does not depend on the manner by which it is connected.

3.6.3 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 3.7 are part of the SOCK SAP C++ wrapper library for sockets [10]. Likewise, the corresponding TLI_* classes are part of the TLI SAP C++ wrapper library for the Transport Layer Interface [6]. SOCK SAP and TLI SAP encapsulate the stream-oriented semantics of connection-oriented protocols like TCP and SPX with a efficient, portable, and type-safe C++ wrappers.

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

• **Concrete Svc Handler:** This class implements the concrete application-specific service activated by a Concrete Connector. A Concrete Svc Handler is instantiated with a specific type of C++ IPC wrapper that exchanges data with its connected peer. The sample code examples in Section 3.7 use a SOCK Stream as the underlying data transport delivery mechanism. It easy to vary the data transfer mechanism, however, by parameterizing the Concrete Svc Handler with a different PEER STREAM (such as a TLI Stream).

• **Concrete Connector:** This class instantiates the generic Connector factory with concrete parameterized type arguments for SVC HANDLER and PEER CONNECTOR. In the sample code in Section 3.7, SOCK Connector is the underlying transport programming interface used to establish a connection actively. However, parameterizing the Connector with a different PEER CONNECTOR (such as a TLI Connector) is straightforward since the IPC mechanisms are encapsulated in C++ wrapper classes. Therefore, the Connector's generic strategy for passively initializing services can be reused, while permitting specific details (such as the underlying network programming interface or the creation strategy) to change flexibly. In particular, note that no Connector components must change when the concurrency strategy is modified.

The following section illustrates sample code that implements the Concrete Svc Handler and Concrete Connector.

3.7 Sample Code

The sample code below illustrates how the Gateway described in Section 2 uses the Connector pattern to simplify the task of actively initializing services by connecting with a large number of Peers. The Gateway plays the active role in establishing connections with Peers (an implementation of a Peer using the Acceptor pattern appears in [1]). Figure 8 illustrates how participants in the Connector pattern are structured in the Gateway.

3.7.1 Svc Handlers for 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 Svc Handler classes inherit from Svc Handler they can be actively connected and initialized by a Connector. To save space, these examples have been simplified by omitting most of the error handling code.

To illustrate the flexibility of the Connector pattern, each open routine in a Svc 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. Note how changes to these

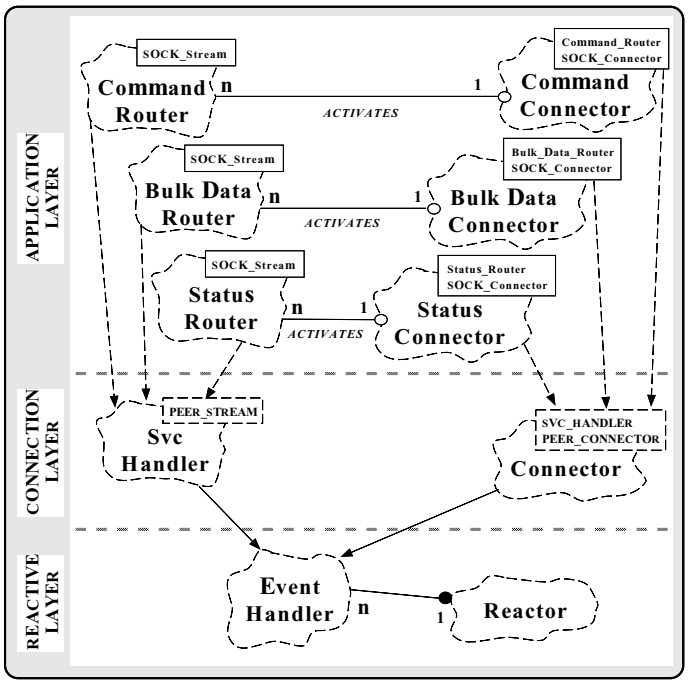

Figure 8: Structure of Participants in the `Gateway` Connector Pattern

concurrency strategies do not affect the architecture of the `Acceptor`, which is generic and thus highly flexible and reusable.

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

```
typedef Svc_Handler <SOCK_Stream> PEER_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 entry point for the thread).
    Thread::spawn (&Status_Router::svc_run, this);
  }

  // Static entry point into the thread, which blocks
  // on the handle_event() call in its own thread.
  static void *svc_run (Status_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` 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 parent process.
      return 0;
    else // 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`:

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

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

3.7.2 The 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.

// Singleton Reactor object.
Reactor reactor;

// Define a Connector factory specialized for
// PEER_ROUTERS.

typedef Connector<PEER_ROUTERS, SOCK_Connector>
```

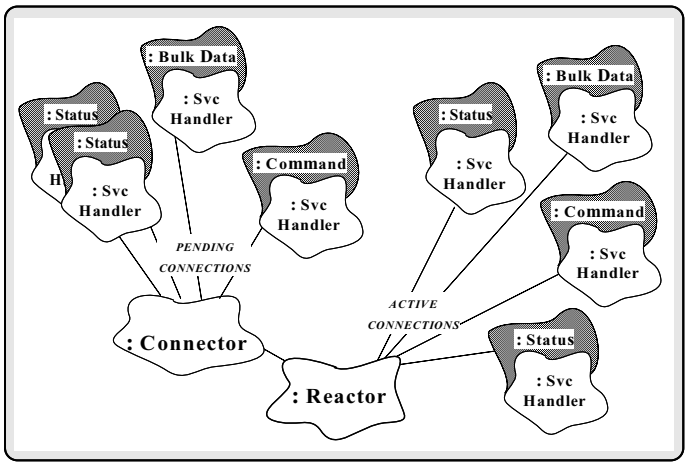

Figure 9: Object Diagram for the `Peer` Connector Pattern

```
        PEER_CONNECTOR;

// 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 (&reactor);

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

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

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

  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.handle_events ();

  /* NOTREACHED */
  return 0;
}
```

All connections are invoked asynchronously. They complete concurrently via `Connector::handle_event` callbacks within the `Reactor`'s event loop, which also demultiplexes and dispatches routing events for `Command_Router` objects. The `Status Routers` and `Bulk Data Routers` execute in separate threads and processes, respectively.

Figure 9 illustrates the relationship between objects in the `Gateway` after four connections have been established.[4] 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`.

3.8 Known Uses

The `Reactor`, `Svc Handler`, and `Connector` classes described in this article are all provided as reusable C++ components in the ACE toolkit [4]. The Connector pattern has been used in the following systems:

- The Ericsson EOS Call Center Management system [11] uses the Connector pattern to allow application-level Call Center Manager `Gateways` to actively establish connections with passive `Peer` hosts in a distributed system.
- The high-speed medical image transfer subsystem of project Spectrum [12] uses the Connector pattern to actively 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.

3.9 Related Patterns

The Connector pattern utilizes several other patterns [2]. It is a Factory that implements a generic Strategy for actively connecting to peers and activating a service handler when the connection is established. The connected service handler then performs its tasks using data exchanged on the connection. Thus, the service is decoupled from the protocol used to establish the connection.

The Connector pattern has an intent similar to the Client/Dispatcher/Server pattern described in [13]. 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 connection establishment.

The Connector pattern is also closely related to the Acceptor pattern, which enables network services to evolve independently of the mechanisms that *passively* establish connections used by the services. These two patterns are duals of each other, in that the Connector handles the "active" side of connection establishment and the Acceptor handles the "passive" side. Thus, the intent, applicability, structure, collaborations, and consequences are very similar.

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

4 Concluding Remarks

This paper describes the Connector pattern and gives an example of how this pattern decouples connection initiation from service initialization and service processing. When used in conjunction with other patterns like the Reactor and Acceptor, this pattern enables the creation of flexible and efficient communication software. UNIX versions of the Connector, Acceptor, and Reactor patterns are freely available via the World Wide Web at URL `http://www.cs.wustl.edu/~schmidt/`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ACE object-oriented network programming toolkit [4] developed at the University of California, Irvine and Washington University. ACE is currently being used in communication software at many companies including Bellcore, Siemens, Motorola, Ericsson, and Kodak.

Thanks to Tim Harrison for comments on this paper.

References

[1] D. C. Schmidt, "Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns," *C++ Report*, vol. 7, November/December 1995.

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

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

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

[5] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

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

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

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

[9] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

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

[11] 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.

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

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