iha.dk

Distributed Real-Time Systems

(TI-DRTS)

POSA2: *Acceptor/Connector Design Pattern*

Version: 7.09.2009

Abstract

The ***Acceptor-Connector*** design pattern decouples the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized

Context

- A networked system or application:
  - in which **connection-oriented** protocols are used to communicate between **peer services**
  - connected via transport endpoints

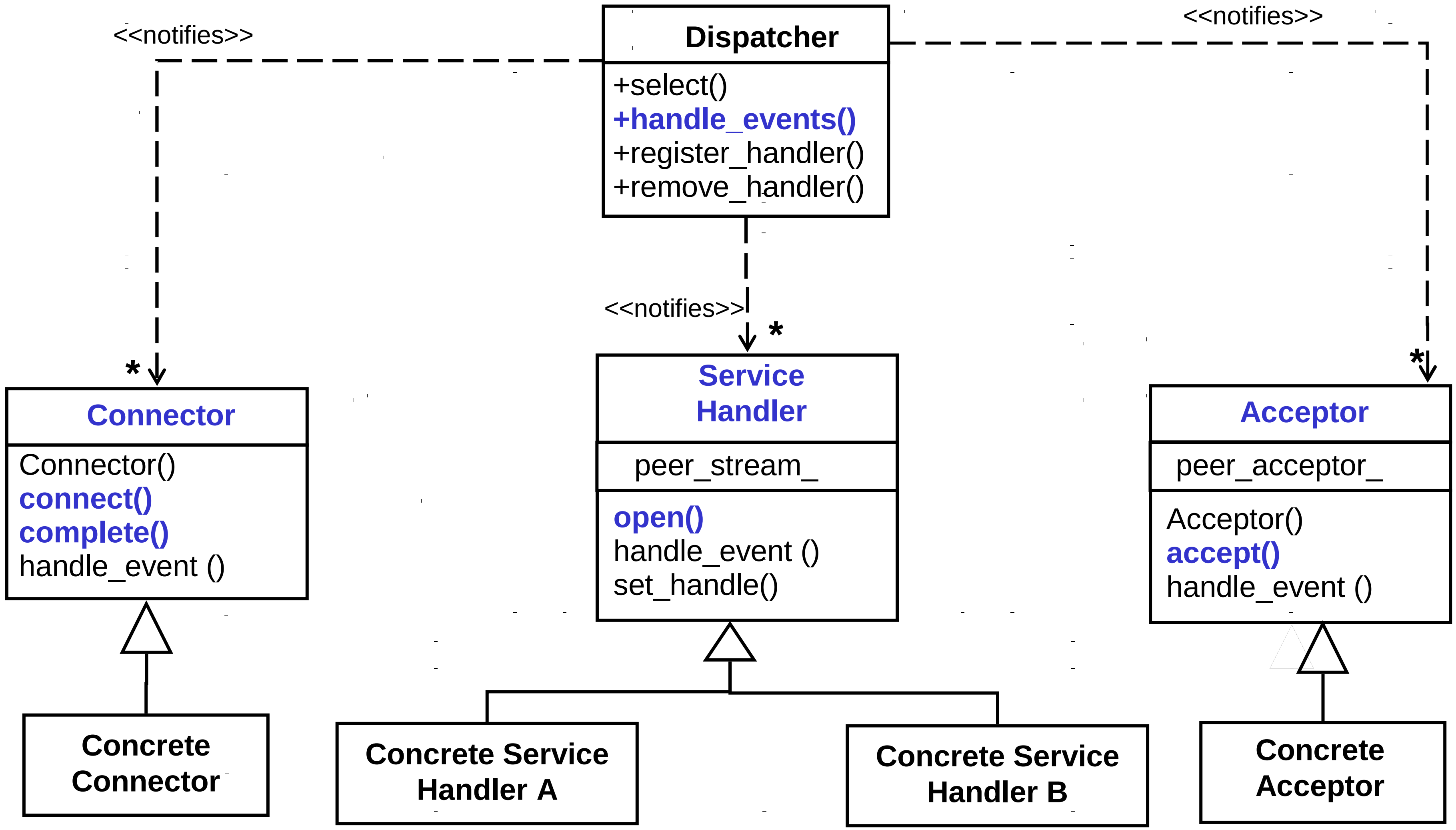
Problem

- Applications in connection-oriented networked systems often contains a significant amount of code that establishes connections and initializes services
- This configuration code is largely independent of the service processing code
  - tightly coupling the configuration code with the service code is undesirable

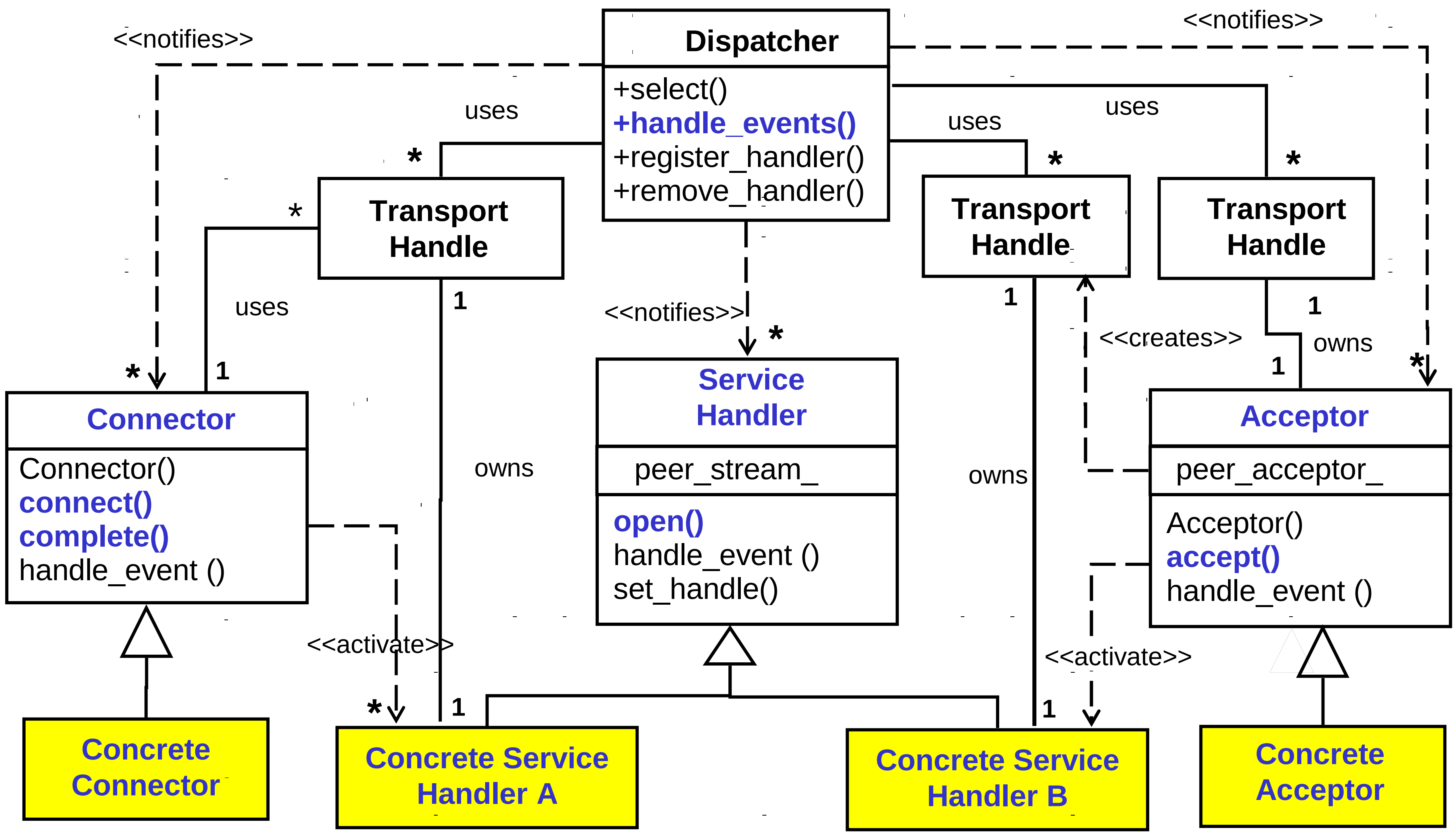
Solution

- Decouple the connection and initialization of peer services from the processing these services perform
- Application services are encapsulated in **Service Handlers**
- Connect and initialize peer service handlers using two factories: **Acceptor** and **Connector**
- Both factories cooperate to create a **full association** between two peer service handlers

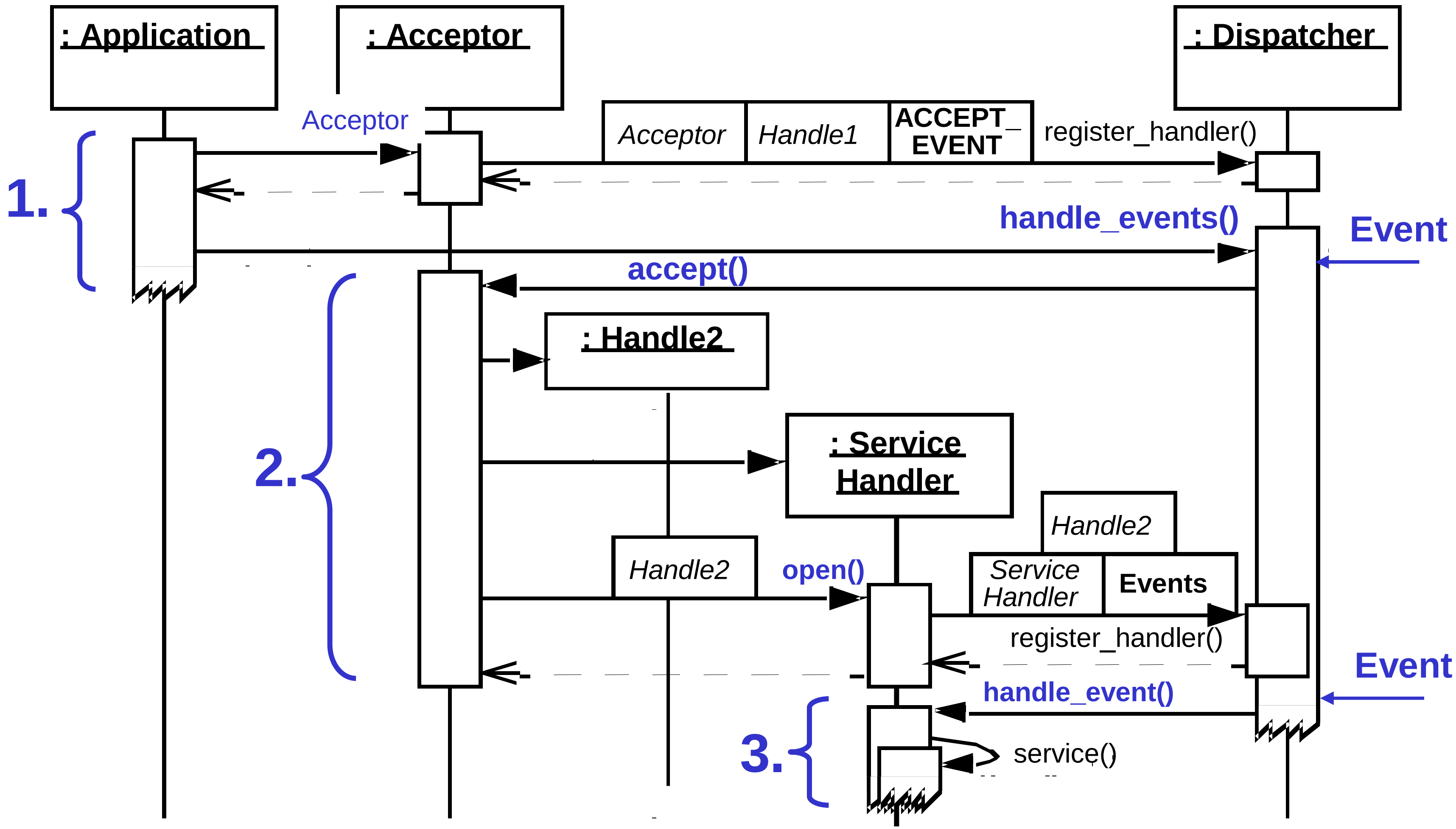
Acceptor/Connector Structure (1)



Acceptor/Connector Structure (2)



Acceptor Dynamics

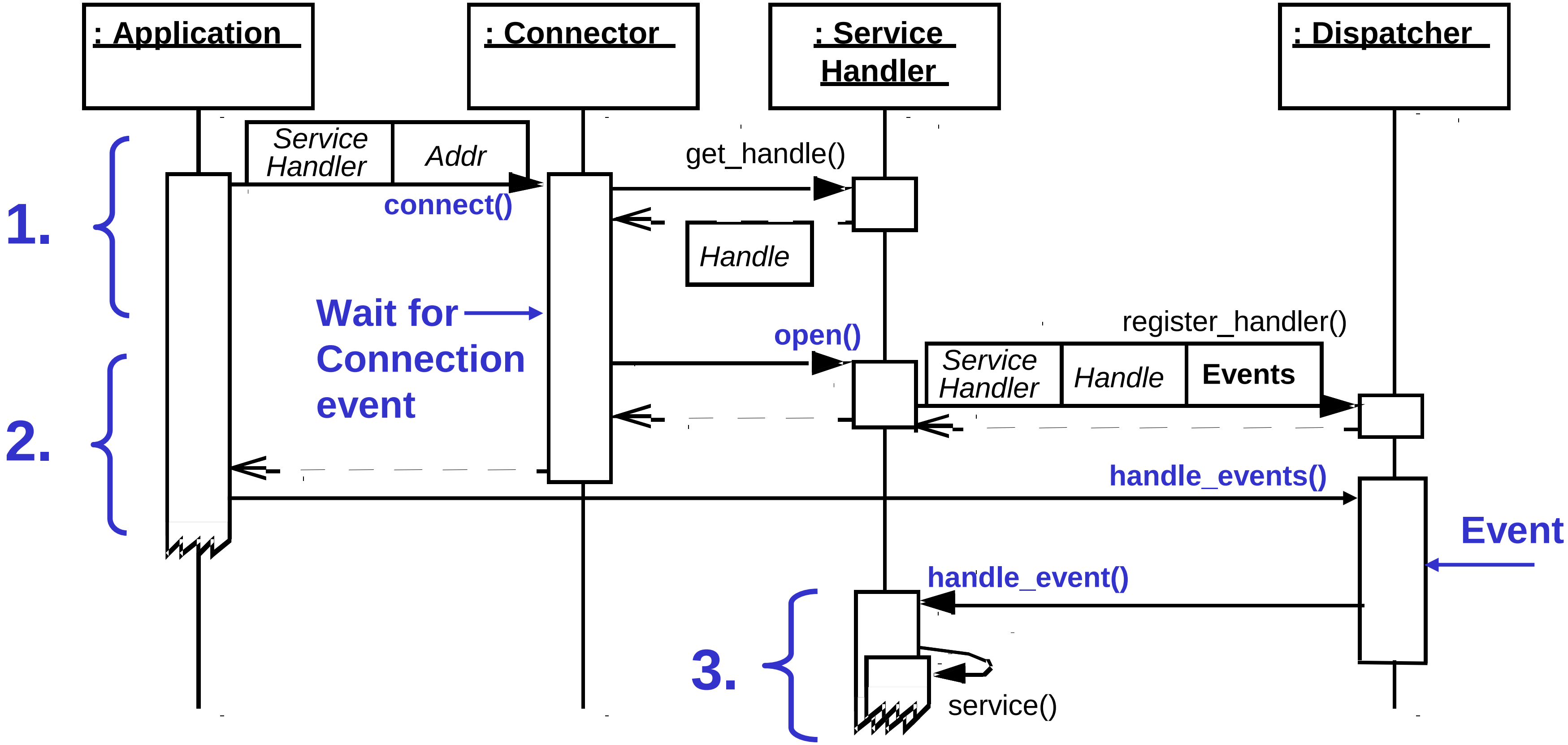


1. Passive-mode endpoint initialize phase
2. Service handler initialize phase
3. Service processing phase

Motivation for Synchrony

- If connection latency is negligible
  - e.g., connecting with a server on the same host via a ‘loopback’ device
- If multiple threads of control are available and it is efficient to use a **thread-per-connection** to connect each service handler synchronously
- If the services must be initialized in a fixed order and the client can’t perform useful work until all connections are established

Synchronous Connector Dynamics

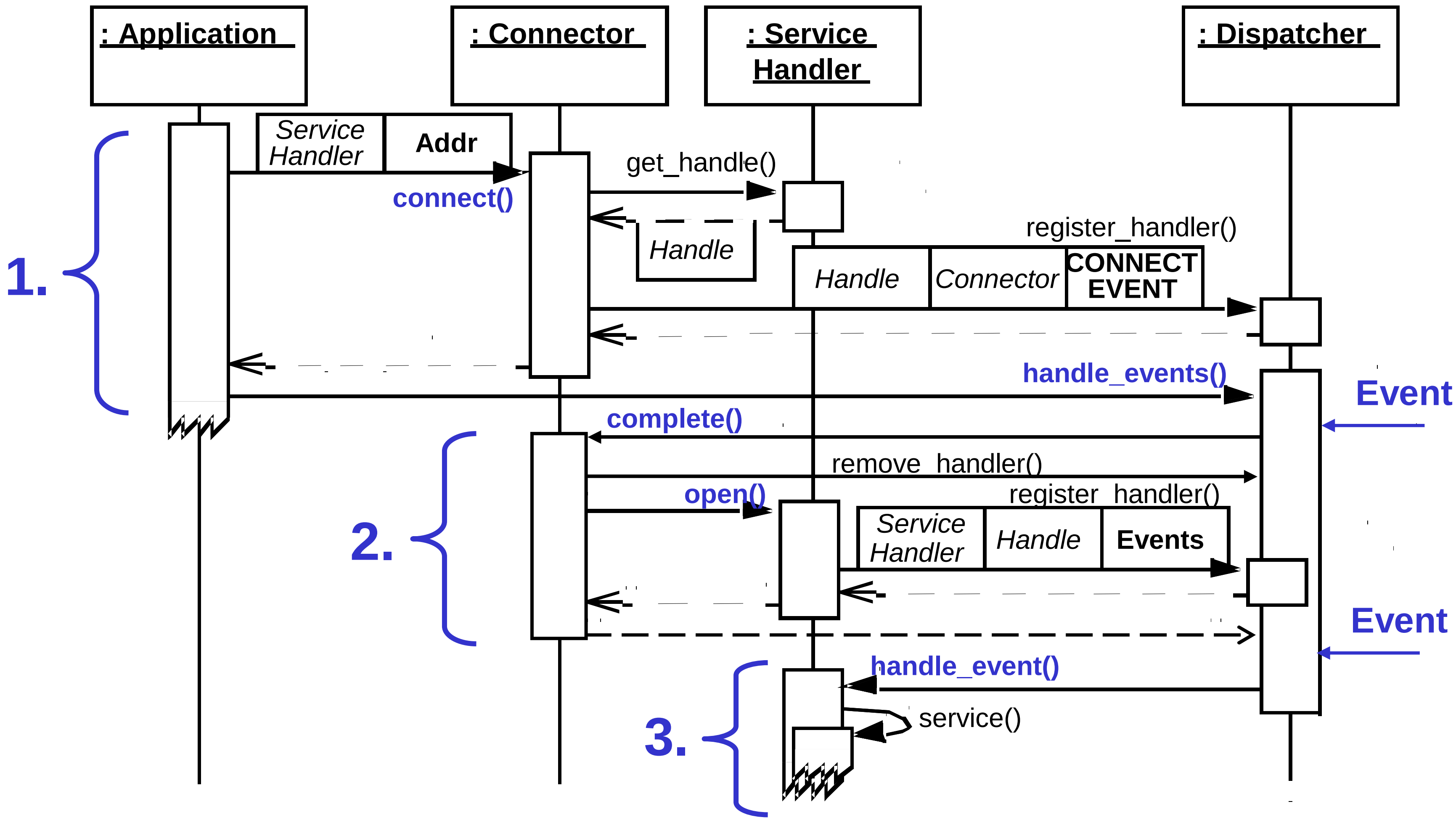


1. *Sync connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*

Motivation for Asynchrony

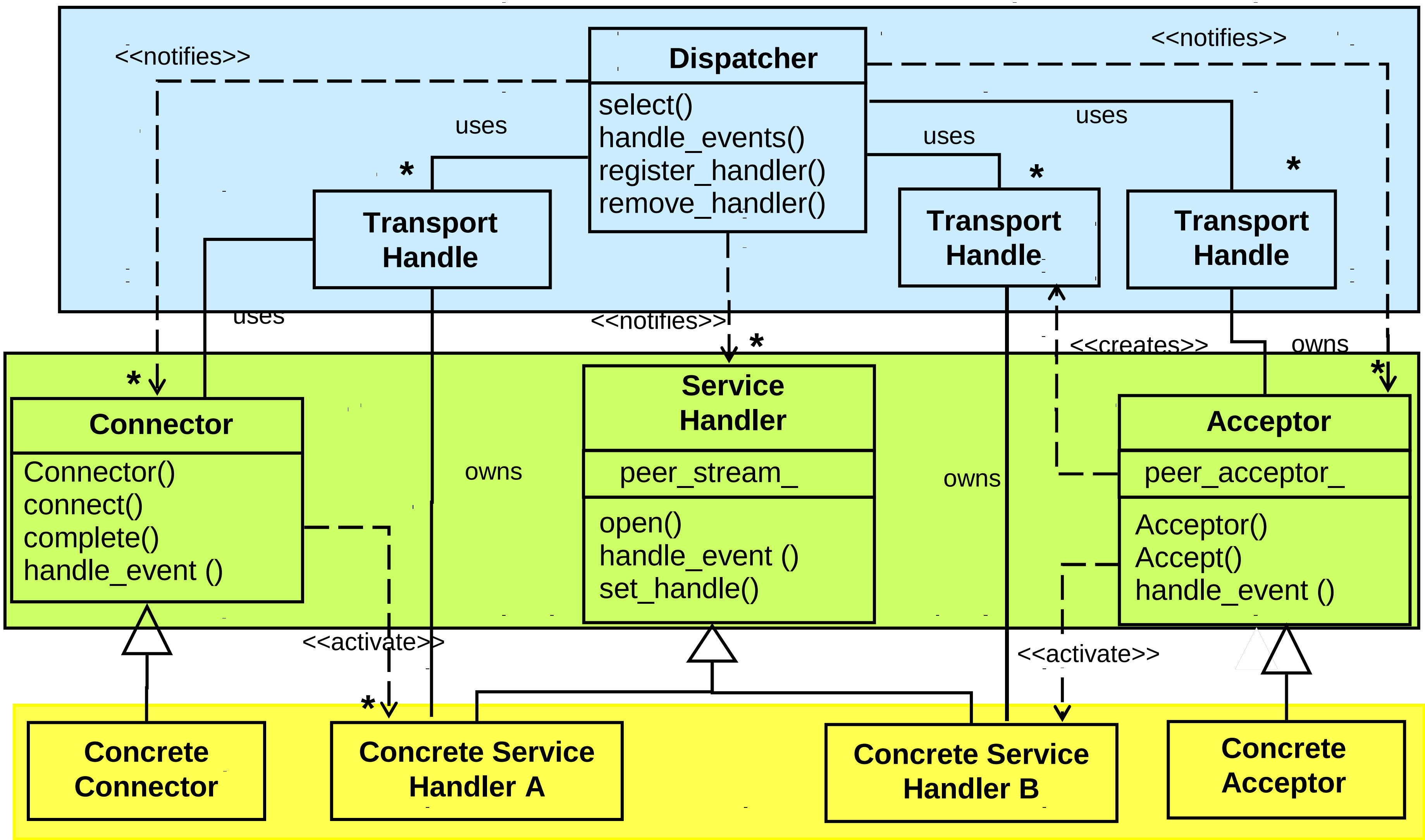
- If client is establishing connections over high latency links
- If client is a single-threaded application
- If client is initializing many peers that can be connected in an arbitrary order

Asynchronous Connector Dynamics



1. *Async connection initiation phase*
2. *Service handler initialize phase*
3. *Service processing phase*

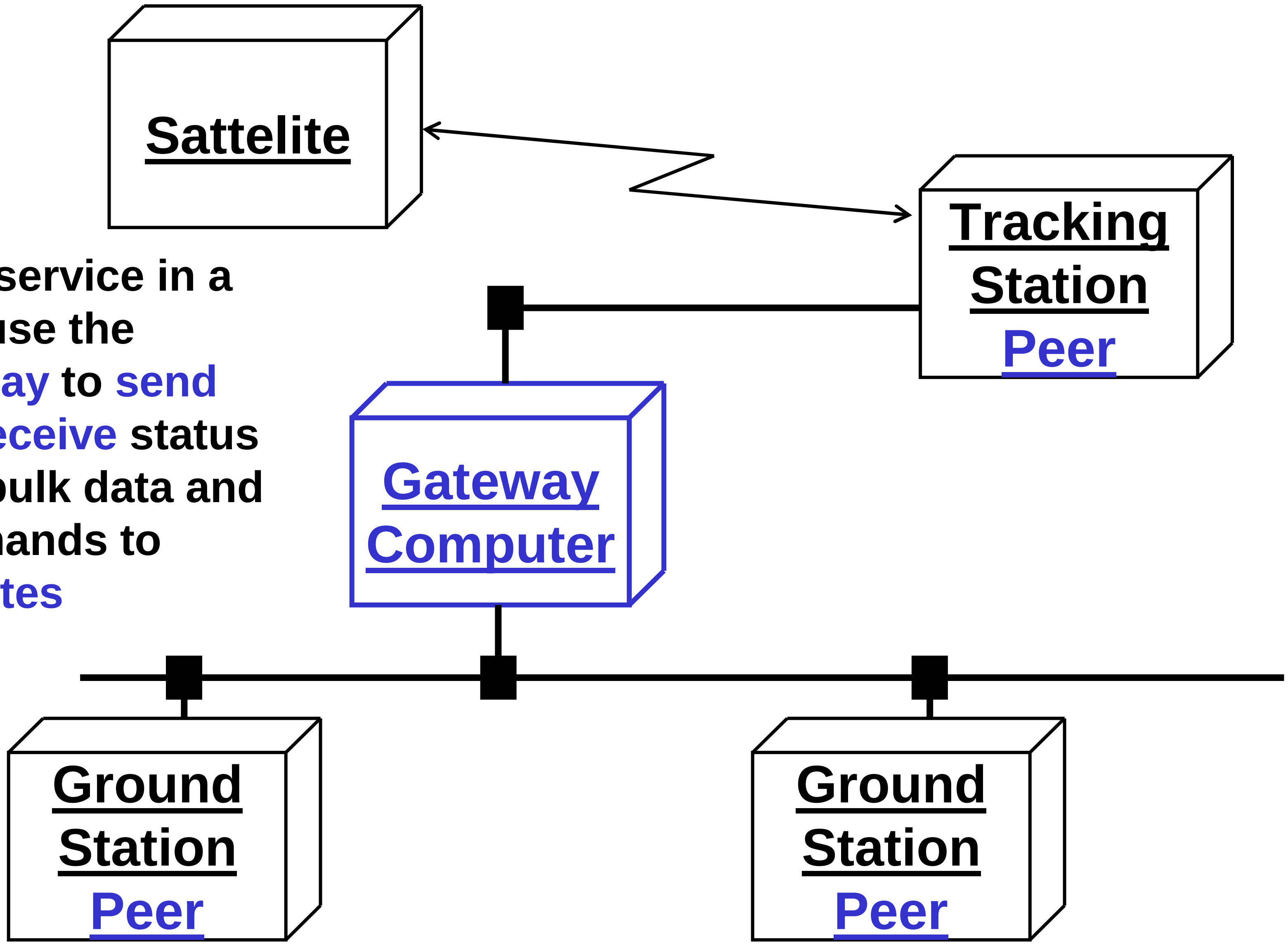
Three Layers



Implementation Steps

1. Implement the ***demultiplexing/ dispatching infrastructure layer*** components
2. Implement the ***connection management layer*** components
3. Implement the ***application layer*** components

Gateway Example



Each service in a peer use the gateway to send and receive status info, bulk data and commands to satellites

1.2. Implement the Dispatching Mechanisms

- A dispatcher is responsible for associating requests to their corresponding acceptors, connectors and service handlers
- Use the ***Reactor*** for synchronous demultiplexing (follow the Reactor guidelines)
- Use the ***Proactor*** for asynchronous demultiplexing (follow the Proactor guidelines)
- Can be implemented as a separate thread using the ***Active Object*** pattern or ***Leader/Followers*** thread pools

Gateway Example

1.1 Select the transport mechanisms

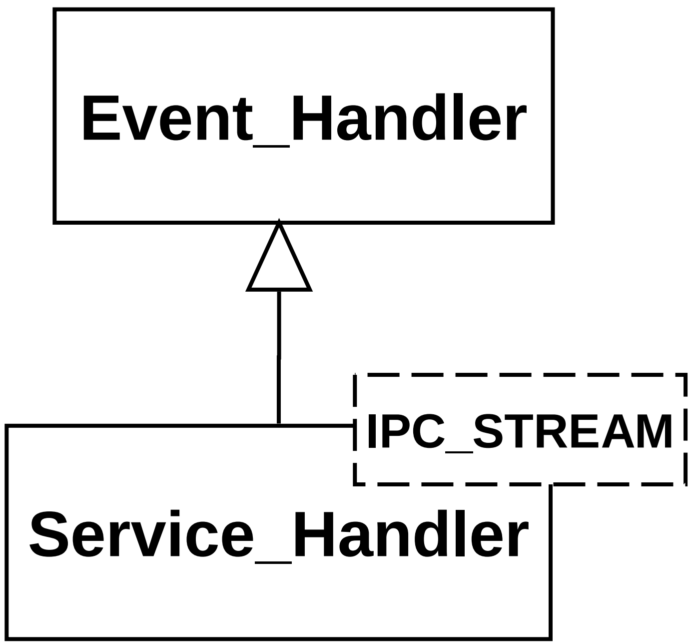
- uses Socket API with passive-mode and data mode sockets
- transport handles = socket handles
- transport address = IP host address and TCP port number

1.2 Implement the dispatching mechanisms

- using components from the Reactor pattern
- within a single thread of control
- using a Singleton Reactor (GoF)

UML Template Class Notation

Generic notation



Concrete Binding to SOCK_Stream Wrapper façade class

Service_Handler Class Definition

```
template <class IPC_STREAM>
class Service_Handler : public Event_Handler {
public:
    typedef typename IPC_STREAM::PEER_ADDR Addr;

    // Hook template method, defined by a subclass
    virtual void open() = 0;
    IPC_STREAM &peer() { return ipc_stream_; }
    Addr &remote_addr()   { return ipc_stream_.remote_addr(); }
    void set_handle(HANDLE handle) {ipc_stream_.set_handle(handle);}
private:
    // Template placeholder for a concrete IPC mechanism wrapper
    // façade, which encapsulate a data-mode transport endpoint and
    // transport handle
    IPC_STREAM ipc_stream_;
};
```

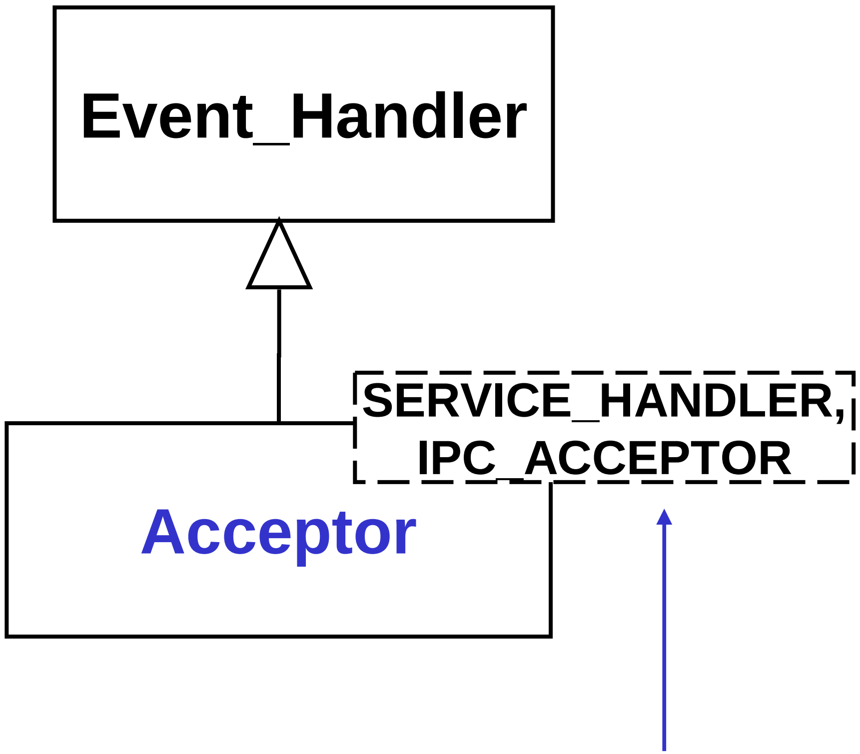
2.2 Define the Generic Acceptor Interface

- A generic acceptor is customized by the components in the application layer
- A customization can be made by using one of two general strategies:
  - **Polymorphism**
    - Using subclassing and a **Template Method** **accept** (GoF)
    - The concrete service handler is created by a **Factory Method** (GoF)
  - **Parameterized types**
    - **accept** is also here a **Template Method** (GoF), which delegates to the IPC mechanism configured into the acceptor class,
    - The service handler can also be supplied as a template parameter

Inheritance - Parameterized Types trade-offs

- Parameterized types may incur additional compile and link-time overhead
  - but generally compile into faster code
- Inheritance may incur additional run-time overhead due to the indirection of dynamic binding
  - but is generally faster to compile and link

Acceptor Template Class



Both of these concrete types are provided by the components in the application layer

Acceptor Class Definition

```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR>
class Acceptor : public Event_Handler {
public:
   typedef typename IPC_ACCEPTOR::PEER_ADDR Addr;
   Acceptor (const Addr &local_addr, Reactor *r);

   virtual void handle_event(HANDLE, Event_Type);
protected:
   virtual void accept();     // template method
   virtual SERVICE_HANDLER *make_service_handler();
   virtual void accept_service_handler(SERVICE_HANDLER *);
   virtual void activate_service_handler(SERVICE_HANDLER *);

   virtual HANDLE get_handle() const;
private:
   IPC_ACCEPTOR peer_acceptor_;     // template placeholder
};
```

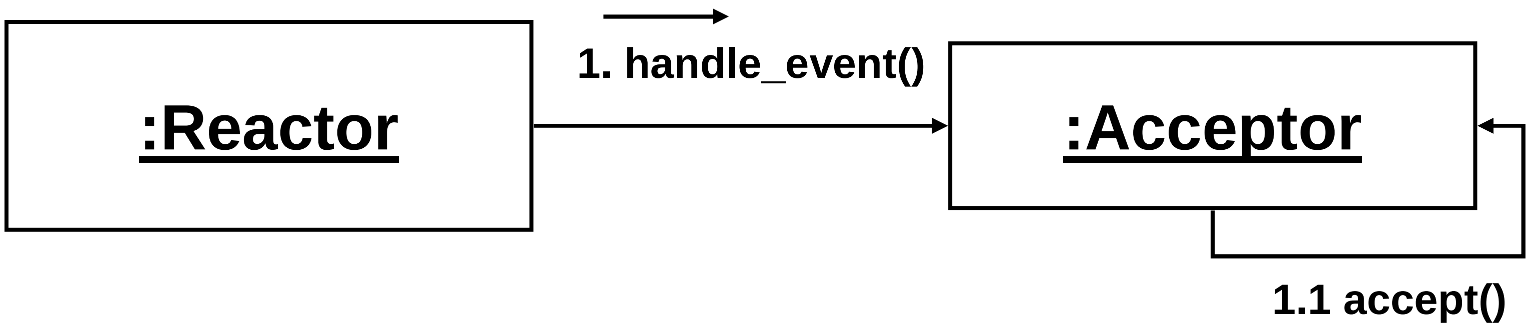
Acceptor Constructor

```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR)
Acceptor<SERVICE_HANDLER, IPC_ACCEPTOR>::Acceptor(
                                        const Addr &local_addr, Reactor *reactor)
{
    // initialize the IPC_ACCEPTOR;
    peer_acceptor_.open(local_addr);

    // register with <reactor>
    reactor->register_handler(this, ACCEPT_MASK);
}
```

Acceptor Event Handling

- In the Gateway Example: the dispatcher is a **Reactor** calling the **handle_event** method defined in the Event_Handler base class and specialized in the Acceptor class



GoF Adapter method

```
Acceptor::handle_event() {
    accept();
}
```

Acceptor::accept Method

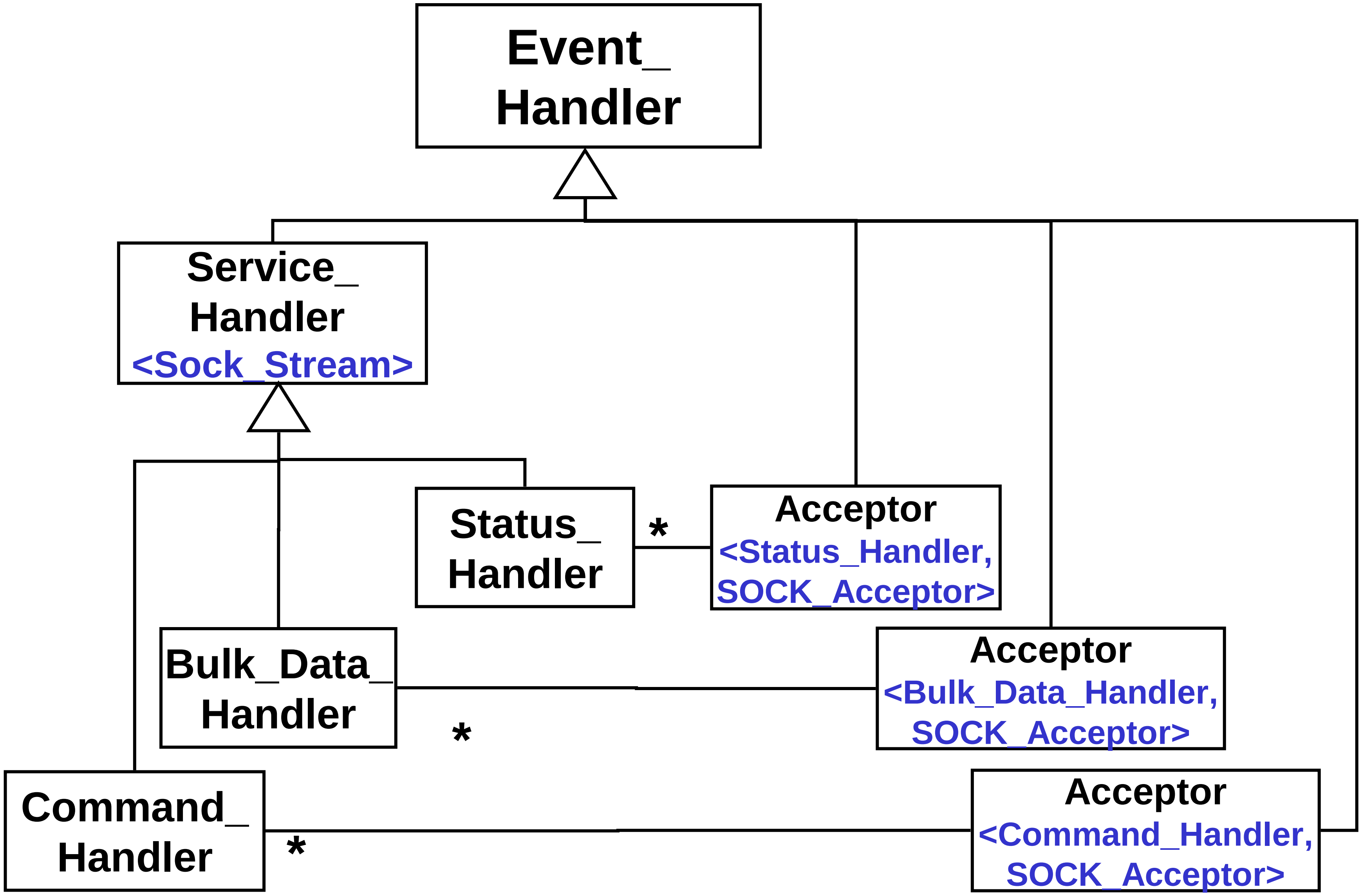
```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR)
void Acceptor<SERVICE_HANDLER, IPC_ACCEPTOR>::accept()
{
   // GoF: Factory Method – creates a new <SERVICE_HANDLER>
   SERVICE_HANDLER *service_handler= make_service_handler();

   // Hook method that accepts a connection passively
   accept_service_handler(service_handler);

   // Hook method that activates the <SERVICE_HANDLER> by
   // invoking its <open> activation hook method
   activate_service_handler(service_handler);
}
```

Example off a GoF Template Method

Peer Host specific Classes



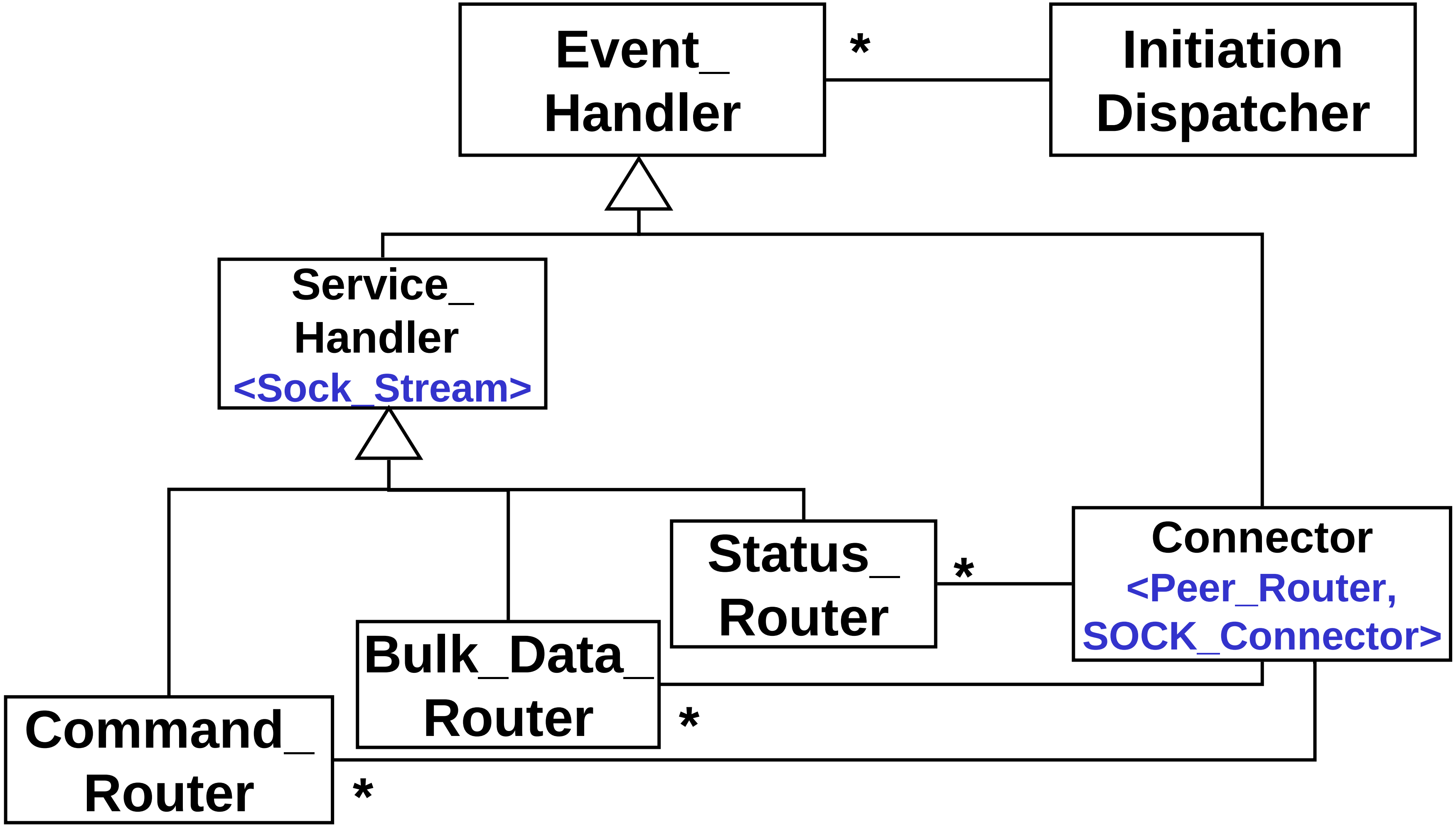
Peer Host Main Program

```
typedef Acceptor<Status_Handler, SOCK_Acceptor> Status_Acceptor;

int main()
{
    // Initialize three concrete acceptors to listen for connections on
    // their well-known ports
    Status_Acceptor s_acceptor(STATUS_PORT, Reactor::instance());
    Bulk_Data_Acceptor bd_acceptor(BULK_DATA_PORT,
                                   Reactor::instance());
    Command_Acceptor c_acceptor(COMMAND_PORT,
                                   Reactor::instance());

    // Event loop that accepts connection request event and processes
    // data from a gateway
    for (; ;)
        Reactor::instance()->handle_events();
}
```

Gateway specific Classes



Gateway Main Program

```
typedef Service_Handler<SOCK_Stream> Peer_Router;
typedef Connector<Peer_Router, SOCK_Connector> Peer_Connector;

int main() {
   Peer_Connector peer_connector(Reactor::instance());
   vector<Peer_Router> peers;

   get_peer_addrs(peers);   // initialize the three routers from a config. file
   typedef vector<Peer_Router>::iterator Peer_Iterator;

   for (Peer_Iterator peer= peers.begin(); peer != peers.end(); peer++) {
      peer_connector.connect( (*peer), peer->remote_addr(),
                                    Peer_Connector::ASYNC);
   }
   for ( ; ;)
      Reactor->instance()->handle_events();
}
```

Acceptor/Connector Benefits

- Reusability, portability, & extensibility
  - This pattern decouples mechanisms for connecting & initializing service handlers from the service processing performed after service handlers are connected & initialized
- Robustness
  - This pattern strongly decouples the service handler from the acceptor, which ensures that a passive-mode transport endpoint can't be used to read or write data accidentally
- Efficiency
  - This pattern can establish connections actively with many hosts asynchronously & efficiently over long-latency wide area networks
  - Asynchrony is important in this situation because a large networked system may have hundreds or thousands of host that must be connected

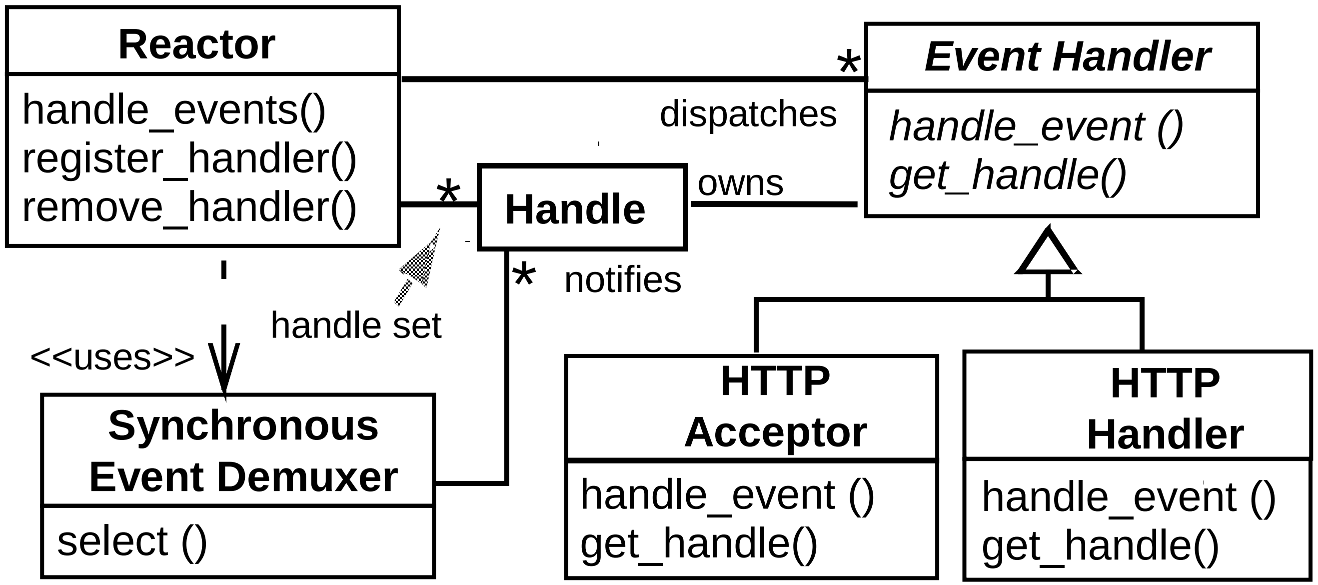
Acceptor/Connecter Liabilities

- Additional indirection
  - The Acceptor-Connector pattern can incur additional indirection compared to using the underlying network programming interfaces directly
- Additional complexity
  - The Acceptor-Connector pattern may add unnecessary complexity for simple client applications that connect with only one server & perform one service using a single network programming interface

Applying the Reactor and Acceptor-Connector Patterns in JAWS

The Reactor architectural pattern decouples:

1. JAWS generic synchronous event demultiplexing & dispatching logic from
2. The HTTP protocol processing it performs in response to events

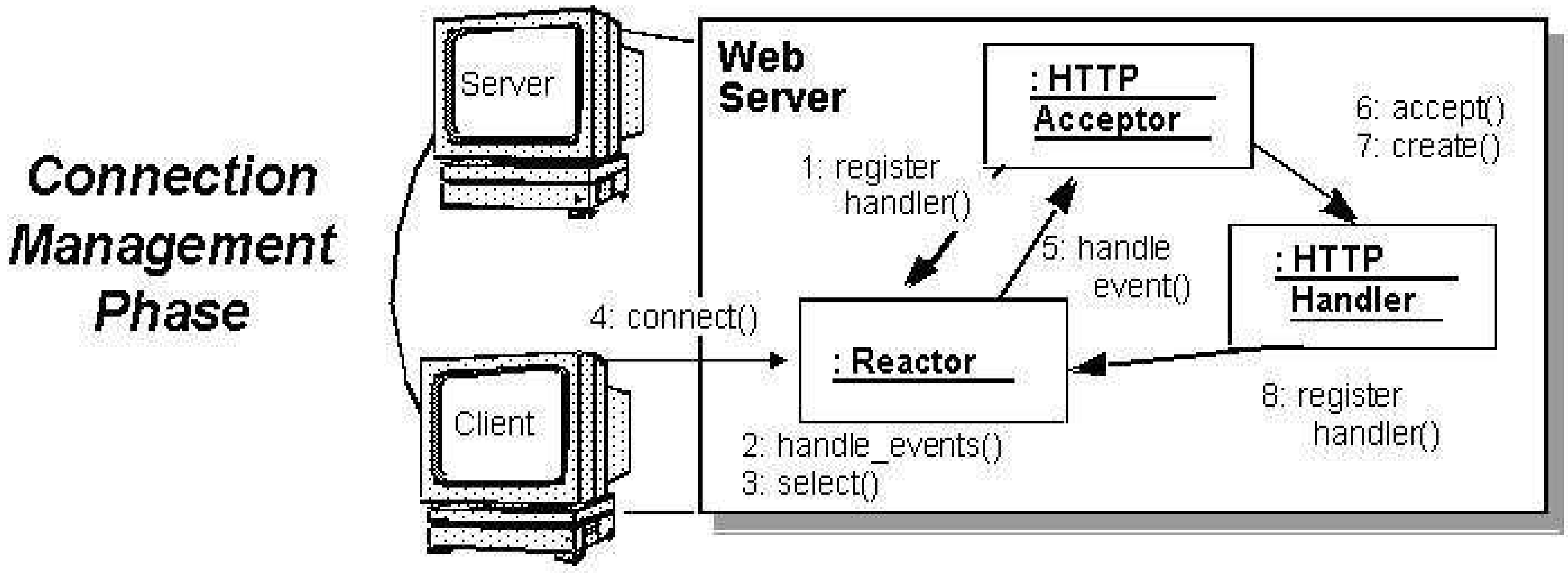


Acceptor/Connector and Reactor

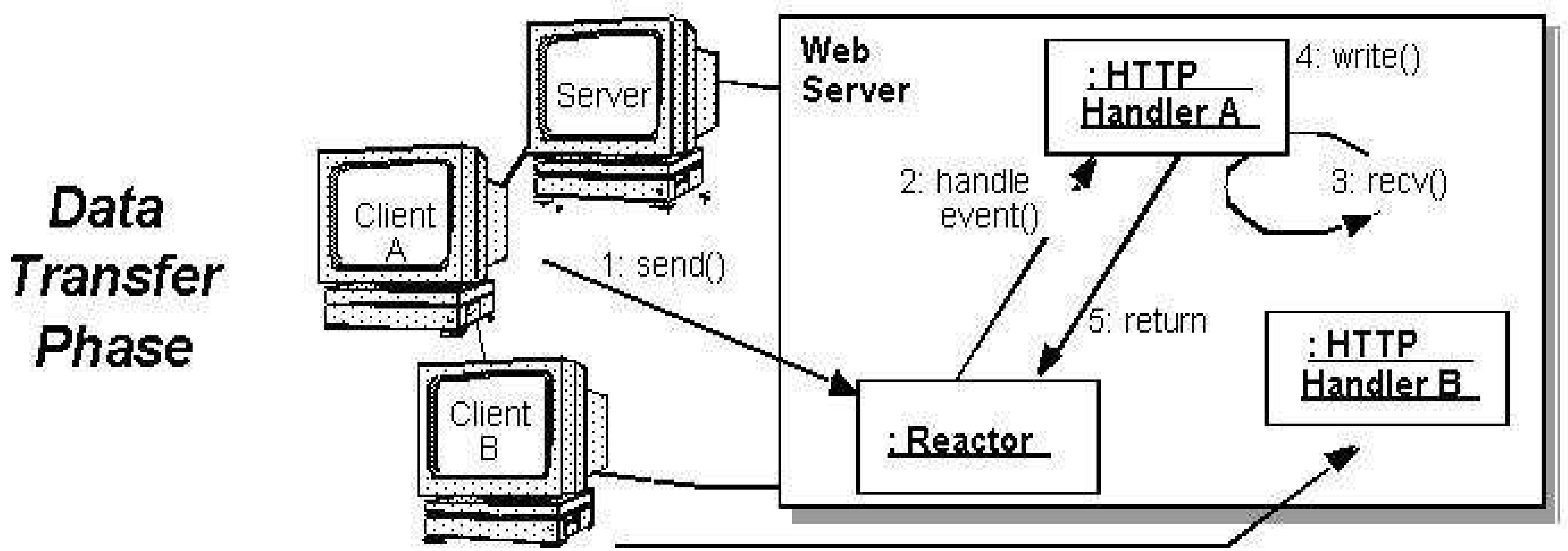
The Acceptor-Connector design pattern can use a Reactor as its *Dispatcher* in order to help decouple:

1. The connection & initialization of peer client & server HTTP services from
2. The processing activities performed by these peer services once they are connected & initialized

Reactive Connection Management in JAWS



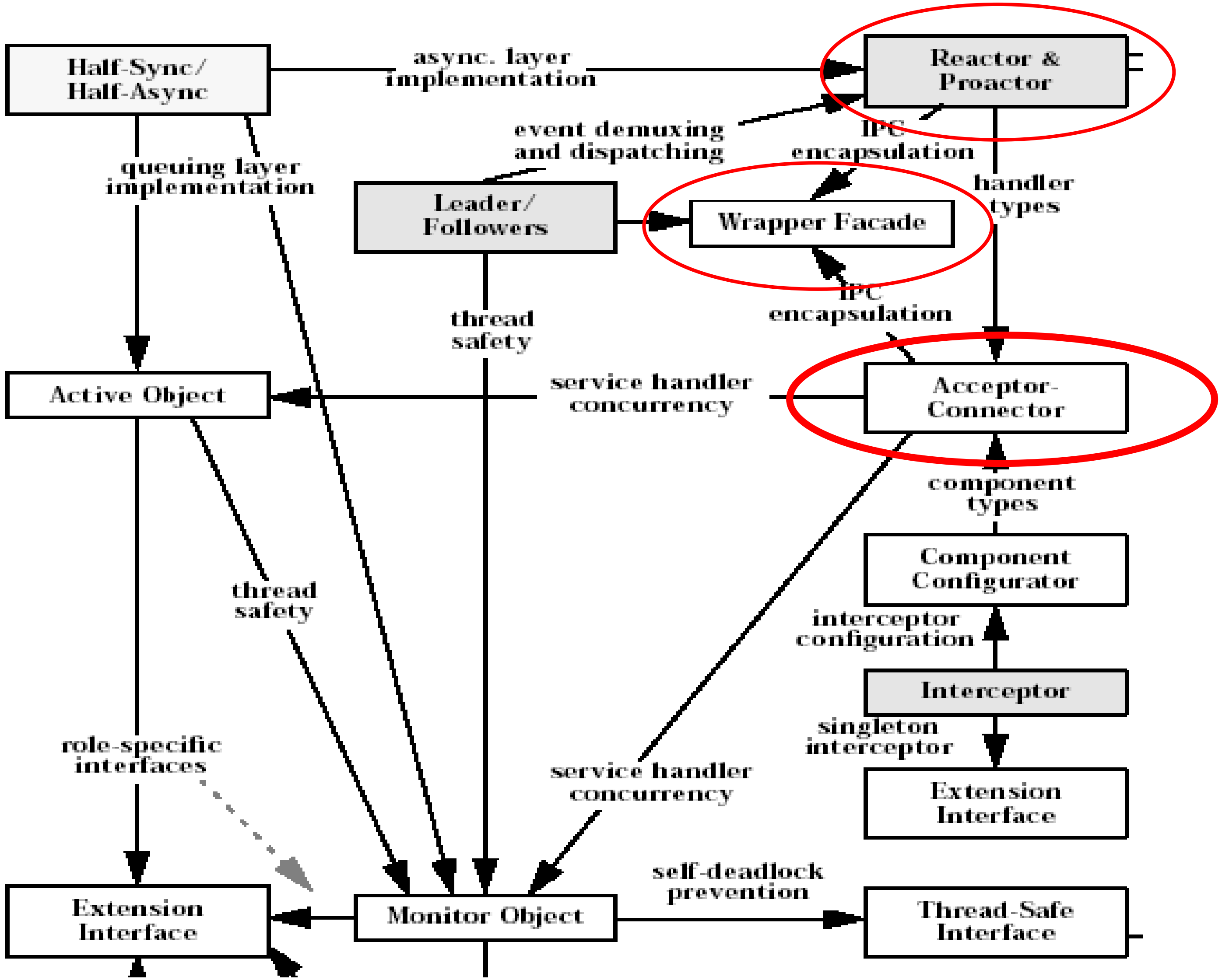
Reactive Data Transfer in JAWS



Known Uses

- Unix network superservers
  - **Inetd**, **Listen**
  - Use a master acceptor process that listen for connections on a set of communication ports – Inetd services: FTP, Telnet, Daytime and Echo
- CORBA Object Request Brokers (ORB)
- Web Browsers
  - **Netscape** + **Internet Explorer** use the asynchronous version of the connector component to establish connections with servers associated with images embedded in HTML pages
- Project Spectrum
  - A high-speed medical image transfer system

Relation to other POSA2 Patterns



Acceptor-Connector – part two

- Connector class
- Concurrency strategies

Connector Class (1)

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
class Connector : public Event_Handler {
public:
    enum Connection_Mode { SYNC, ASYNC };
    typedef typename IPC_CONNECTOR::PEER_ADDR Addr;
    Connector(Reactor *reactor): reactor_(reactor) { }
                        // Template Method
    void connect( SERVICE_HANDLER *sh, const Addr &remote_addr,
                        Connection_Mode mode) {
            connect_service_handler(sh, remote_addr, mode);
    }
                        // Adapter Method
    virtual void handle_event(HANDLE handle, Event_Type) {
            complete(handle);
    }
protected:
    virtual void complete(HANDLE handle);
```

Connector Class (2)

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
class Connector : public Event_Handler {
        // Continued from previous slide
  virtual void connect_service_handler(const Addr &addr,
                                       Connection_Mode mode);
  int register_handler(SERVICE_HANDLER *sh,
                       Connection_Mode mode);
  virtual void activate_service_handler(SERVICE_HANDLER *sh);
private:
  IPC_CONNECTOR connector_;
  // C++ standard library map that associates <HANDLES> with
  // <SERVICE_HANDLER> s for pending connections
  typedef map<HANDLE, SERVICE_HANDLER*> Connection_Map;
  Connection_Map connection_map_;
  Reactor *reactor_;
}
```

Connector::connect_service_handler()

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
class Connector< SERVICE_HANDLER, IPC_CONNECTOR>::
connect_service_handler(SERVICE_HANDLER *svc_handler,
                        const Addr &addr, Connection_Mode mode) {
  try {
      connector_.connect(*svc_handler, addr, mode);
      // activate if we connect synchronously
      activate_service_handler(svc_handler);
  } catch (SystemEx &ex)
      if (ex.status() == EWOULDBLOCK && mode == ASYNC) {
        // register for asynchronously call back
        reactor_->register_handler(this,WRITE_MASK);
        // store <service handler *> in map
        connection_map_[connector_.get_handle()]= svc_handler;
      }
  }
}
```

Connector::complete()

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
class Connector< SERVICE_HANDLER, IPC_CONNECTOR>::complete(
                                                  HANDLE handle) {
   Connection_Map ::iterator i = connection_map_.find(handle);
   if (i == connection_map_.end() ) throw  ….

   // we just want the value part of the <key,value>
   SERVICE_HANDLER *svc_handler = (*i).second;
   // Transfer I/O Handle to <service_handler>
   svc_handler->set_handle(handle);
   reactor_->remove_handler(handle, WRITE_MASK);
   connection_map_.erase(i);

   // connection is complete so activate handler
   activate_service_handler(svc_handler);
}
```

Examples of Flexibility

- Each concrete service handler could implement a different concurrency strategy
  - Status_Router can run in a separate thread
  - Bulk_Data_Router as a separate process
  - Command_Router runs in the same thread as the Reactor
- The concurrency strategy is implemented in the **open()** hook method

Status_Router Class

```
class Status_Router: public Peer_Router {
public:
    virtual void open() {
        // Make this handler run in separate thread
        Thread_Manager::instance()->spawn(
            &Status_Handler::svc_run, this);
    }
    static void *svc_run(Status_Handler *this_obj) {
        for (; ;)
            this_obj->run();
    }
    // receive and process status data from/to peers
    virtual void run() {
            char buf[BUFSIZ];
            peer().recv(buf, sizeof(buf));
            // routing takes place here
    }
};
```

Bulk_Data_Router Class

```
class Bulk_Data_Router: public Peer_Router {
public:
    virtual void open() {
        // activate router in a separate process
        if (fork() ==0) {
            // this method can block because it runs in its own process
            for (; ;)
                run();
        }

        // receive and route bulk data from/to peers
        virtual void run() {
            char buf[BUFSIZ];
            peer().recv(buf, sizeof(buf));
            // routing takes place here
        }
}
```