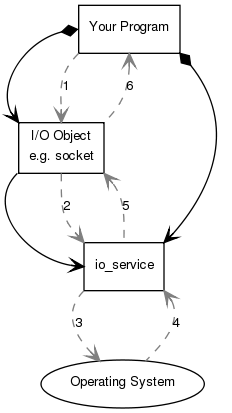
#### [Basic Boost.Asio Anatomy](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/basics.html)

Boost.Asio may be used to perform both synchronous and asynchronous operations on I/O objects such as sockets. Before using Boost.Asio it may be useful to get a conceptual picture of the various parts of Boost.Asio, your program, and how they work together.

As an introductory example, let's consider what happens when you perform a connect operation on a socket. We shall start by examining synchronous operations.



**Your program** will have at least one **io\_service** object. The **io\_service** represents **your program**'s link to the **operating system**'s I/O services.

boost::asio::io\_service io\_service;

To perform I/O operations **your program** will need an **I/O object** such as a TCP socket:

boost::asio::ip::tcp::socket socket(io\_service);

When a synchronous connect operation is performed, the following sequence of events occurs:

1. **Your program** initiates the connect operation by calling the **I/O object**:

socket.connect(server\_endpoint);

2. The **I/O object** forwards the request to the **io\_service**.

3. The **io\_service** calls on the **operating system** to perform the connect operation.

4. The **operating system** returns the result of the operation to the **io\_service**.

5. The **io\_service** translates any error resulting from the operation into a boost::system::error\_code. An error\_code may be compared with specific values, or tested as a boolean (where a falseresult means that no error occurred). The result is then forwarded back up to the **I/O object**.

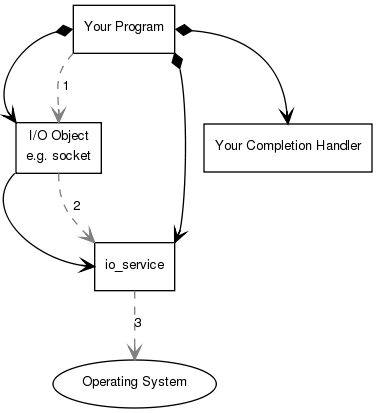
6. The **I/O object** throws an exception of type boost::system::system\_error if the operation failed. If the code to initiate the operation had instead been written as:

boost::system::error\_code ec;

socket.connect(server\_endpoint, ec);

then the error\_code variable ec would be set to the result of the operation, and no exception would be thrown.

When an asynchronous operation is used, a different sequence of events occurs.



1. **Your program** initiates the connect operation by calling the **I/O object**:

socket.async\_connect(server\_endpoint, your\_completion\_handler);

where your\_completion\_handler is a function or function object with the signature:

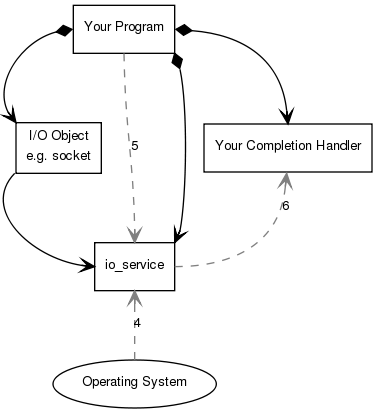
void your\_completion\_handler(const boost::system::error\_code& ec);

The exact signature required depends on the asynchronous operation being performed. The reference documentation indicates the appropriate form for each operation.

2. The **I/O object** forwards the request to the **io\_service**.

3. The **io\_service** signals to the **operating system** that it should start an asynchronous connect.

Time passes. (In the synchronous case this wait would have been contained entirely within the duration of the connect operation.)



4. The **operating system** indicates that the connect operation has completed by placing the result on a queue, ready to be picked up by the **io\_service**.

5. **Your program** must make a call to io\_service::run() (or to one of the similar **io\_service** member functions) in order for the result to be retrieved. A call to io\_service::run() blocks while there are unfinished asynchronous operations, so you would typically call it as soon as you have started your first asynchronous operation.

6. While inside the call to io\_service::run(), the **io\_service** dequeues the result of the operation, translates it into an error\_code, and then passes it to **your completion handler**.

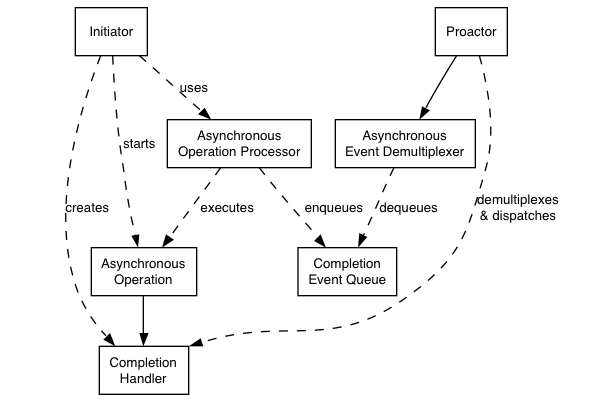
This is a simplified picture of how Boost.Asio operates. You will want to delve further into the documentation if your needs are more advanced, such as extending Boost.Asio to perform other types of asynchronous operations.

#### [The Proactor Design Pattern: Concurrency Without Threads](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html)

The Boost.Asio library offers side-by-side support for synchronous and asynchronous operations. The asynchronous support is based on the Proactor design pattern [[POSA2]](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html#boost_asio.overview.core.async.references). The advantages and disadvantages of this approach, when compared to a synchronous-only or Reactor approach, are outlined below.

###### [Proactor and Boost.Asio](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.proactor_and_boost_asio)

Let us examine how the Proactor design pattern is implemented in Boost.Asio, without reference to platform-specific details.



**Proactor design pattern (adapted from [POSA2])**

— Asynchronous Operation

Defines an operation that is executed asynchronously, such as an asynchronous read or write on a socket.

— Asynchronous Operation Processor

Executes asynchronous operations and queues events on a completion event queue when operations complete. From a high-level point of view, services like stream\_socket\_service are asynchronous operation processors.

— Completion Event Queue

Buffers completion events until they are dequeued by an asynchronous event demultiplexer.

— Completion Handler

Processes the result of an asynchronous operation. These are function objects, often created using boost::bind.

— Asynchronous Event Demultiplexer

Blocks waiting for events to occur on the completion event queue, and returns a completed event to its caller.

— Proactor

Calls the asynchronous event demultiplexer to dequeue events, and dispatches the completion handler (i.e. invokes the function object) associated with the event. This abstraction is represented by the io\_service class.

— Initiator

Application-specific code that starts asynchronous operations. The initiator interacts with an asynchronous operation processor via a high-level interface such as basic\_stream\_socket, which in turn delegates to a service like stream\_socket\_service.

###### [Implementation Using Reactor](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.implementation_using_reactor)

On many platforms, Boost.Asio implements the Proactor design pattern in terms of a Reactor, such as select, epoll or kqueue. This implementation approach corresponds to the Proactor design pattern as follows:

— Asynchronous Operation Processor

A reactor implemented using select, epoll or kqueue. When the reactor indicates that the resource is ready to perform the operation, the processor executes the asynchronous operation and enqueues the associated completion handler on the completion event queue.

— Completion Event Queue

A linked list of completion handlers (i.e. function objects).

— Asynchronous Event Demultiplexer

This is implemented by waiting on an event or condition variable until a completion handler is available in the completion event queue.

###### [Implementation Using Windows Overlapped I/O](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.implementation_using_windows_overlapped_i_o)

On Windows NT, 2000 and XP, Boost.Asio takes advantage of overlapped I/O to provide an efficient implementation of the Proactor design pattern. This implementation approach corresponds to the Proactor design pattern as follows:

— Asynchronous Operation Processor

This is implemented by the operating system. Operations are initiated by calling an overlapped function such as AcceptEx.

— Completion Event Queue

This is implemented by the operating system, and is associated with an I/O completion port. There is one I/O completion port for each io\_service instance.

— Asynchronous Event Demultiplexer

Called by Boost.Asio to dequeue events and their associated completion handlers.

###### [Advantages](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.advantages)

— Portability.

Many operating systems offer a native asynchronous I/O API (such as overlapped I/O on Windows) as the preferred option for developing high performance network applications. The library may be implemented in terms of native asynchronous I/O. However, if native support is not available, the library may also be implemented using synchronous event demultiplexors that typify the Reactor pattern, such as POSIX select().

— Decoupling threading from concurrency.

Long-duration operations are performed asynchronously by the implementation on behalf of the application. Consequently applications do not need to spawn many threads in order to increase concurrency.

— Performance and scalability.

Implementation strategies such as thread-per-connection (which a synchronous-only approach would require) can degrade system performance, due to increased context switching, synchronisation and data movement among CPUs. With asynchronous operations it is possible to avoid the cost of context switching by minimising the number of operating system threads — typically a limited resource — and only activating the logical threads of control that have events to process.

— Simplified application synchronisation.

Asynchronous operation completion handlers can be written as though they exist in a single-threaded environment, and so application logic can be developed with little or no concern for synchronisation issues.

— Function composition.

Function composition refers to the implementation of functions to provide a higher-level operation, such as sending a message in a particular format. Each function is implemented in terms of multiple calls to lower-level read or write operations.

For example, consider a protocol where each message consists of a fixed-length header followed by a variable length body, where the length of the body is specified in the header. A hypothetical read\_message operation could be implemented using two lower-level reads, the first to receive the header and, once the length is known, the second to receive the body.

To compose functions in an asynchronous model, asynchronous operations can be chained together. That is, a completion handler for one operation can initiate the next. Starting the first call in the chain can be encapsulated so that the caller need not be aware that the higher-level operation is implemented as a chain of asynchronous operations.

The ability to compose new operations in this way simplifies the development of higher levels of abstraction above a networking library, such as functions to support a specific protocol.

###### [Disadvantages](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.disadvantages)

— Program complexity.

It is more difficult to develop applications using asynchronous mechanisms due to the separation in time and space between operation initiation and completion. Applications may also be harder to debug due to the inverted flow of control.

— Memory usage.

Buffer space must be committed for the duration of a read or write operation, which may continue indefinitely, and a separate buffer is required for each concurrent operation. The Reactor pattern, on the other hand, does not require buffer space until a socket is ready for reading or writing.

###### [References](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.references)

[POSA2] D. Schmidt et al, Pattern Oriented Software Architecture, Volume 2. Wiley, 2000.

#### [Threads and Boost.Asio](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/threads.html)

###### [Thread Safety](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/threads.html" \l "boost_asio.overview.core.threads.thread_safety)

In general, it is safe to make concurrent use of distinct objects, but unsafe to make concurrent use of a single object. However, types such as io\_service provide a stronger guarantee that it is safe to use a single object concurrently.

###### [Thread Pools](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/threads.html" \l "boost_asio.overview.core.threads.thread_pools)

Multiple threads may call io\_service::run() to set up a pool of threads from which completion handlers may be invoked. This approach may also be used with io\_service::post() to use a means to perform any computational tasks across a thread pool.

Note that all threads that have joined an io\_service's pool are considered equivalent, and the io\_service may distribute work across them in an arbitrary fashion.

###### [Internal Threads](http://67.223.234.84/boost_doc/doc/html/boost_asio/overview/core/threads.html" \l "boost_asio.overview.core.threads.internal_threads)

The implementation of this library for a particular platform may make use of one or more internal threads to emulate asynchronicity. As far as possible, these threads must be invisible to the library user. In particular, the threads:

* must not call the user's code directly; and
* must block all signals.

|  |  |
| --- | --- |
| [Note] | Note |
| The implementation currently violates the first of these rules for the following functions:  — ip::basic\_resolver::async\_resolve() on all platforms.  — basic\_socket::async\_connect() on Windows.  — Any operation involving null\_buffers() on Windows, other than an asynchronous read performed on a stream-oriented socket. |

This approach is complemented by the following guarantee:

* Asynchronous completion handlers will only be called from threads that are currently calling io\_service::run().

Consequently, it is the library user's responsibility to create and manage all threads to which the notifications will be delivered.

The reasons for this approach include:

* By only calling io\_service::run() from a single thread, the user's code can avoid the development complexity associated with synchronisation. For example, a library user can implement scalable servers that are single-threaded (from the user's point of view).
* A library user may need to perform initialisation in a thread shortly after the thread starts and before any other application code is executed. For example, users of Microsoft's COM must callCoInitializeEx before any other COM operations can be called from that thread.
* The library interface is decoupled from interfaces for thread creation and management, and permits implementations on platforms where threads are not available.