Labor-Saving Architecture:
An Object-Oriented Framework for Networked Software

William R. Otte and Douglas C. Schmidt
Vanderbilt University, Nashville, TN

Developing software for networked applications is hard and developing reusable software for networked applications is even harder. First, there are the complexities inherent to distributed systems, such as optimally mapping application services onto hardware nodes, synchronizing service initialization, and ensuring availability while masking partial failures. These complexities can stymie even experienced software developers because they arise from fundamental challenges in the domain of network programming.

Unfortunately, developers must master the *accidental complexities*, such as low-level and non-portable programming interfaces and the use of function-oriented design techniques that require tedious and error-prone revisions as requirements and/or platforms evolve. These complexities arise largely from limitations with the software tools and techniques applied historically by developers of networked software.

Despite the use of object-oriented technologies in many domains, such as graphical user interfaces and productivity tools, much networked software still uses C-level operating system (OS) application programmatic interfaces (APIs), such as the UNIX socket API or the Windows threading API. Many accidental complexities of networked programming stem from the use of these C-level OS APIs, which are not type-safe, often not reentrant, and are not portable across OS platforms. The C APIs were also designed before the widespread adoption of modern design methods and technologies, so they encourage developers to decompose their problems functionally in terms of processing steps in a top-down design, instead of using OO design and programming techniques. Experience over the past several decades has shown that functional decomposition of non-trivial software complicates maintenance and evolution because functional requirements are rarely stable design centers [OOSC].

Fortunately, two decades of advances in design/implementation techniques and programming languages have made it much easier to write and reuse networked software. In particular, object-oriented (OO) programming languages (such as C++, Java, and C#) combined with *patterns* (such as Wrapper Facades [POSA2], Adapters, and Template Method [GoF]) and frameworks (such as host infrastructure middleware like ACE [C++NPv2] and the Java class libraries for network programming [JNP], and similar host infrastructure middleware) help to encapsulate low-level functional OS APIs and mask syntactic and semantic differences between platforms. As a result, developers can focus on application-specific behavior and properties in their software, rather than repeatedly wrestling with the accidental complexities of programming the low-level networking and OS infrastructure.

A key benefit of applying patterns and frameworks to networked software is that they can help developers help craft reusable architectures that (1) capture the common structure and behavior in a particular domain and (2) make it easy to change or replace various algorithms, policies, and mechanisms selectively without affecting other existing parts of the architecture. While most developers of networked software can apply well-designed

OO frameworks to their applications, the knowledge of how to create such a framework remains a black art that has historically been learned only by extensive (and expensive) trial and error. In addition to the conventional challenges of devising a flexible OO design that can expand and contract to meet new requirements, networked software must often run efficiently and scalably in a range of operating environments. The goal of this chapter is to help demystify the black art of OO frameworks for networked software by using a case study to systematically dissect the design and implementation of a representative networked software application.

In general, the beauty of our solution stems from its use of patterns and OO techniques to balance key domain forces, such as reusability, extensibility, and performance. In particular, our approach enables developers to identify common design/programming artifacts, thereby enhancing reuse. It also provide a means to encapsulate variabilities in a common and parameterizable way, thereby enhancing extensibility and portability.

Sample Application: Logging Service

The OO software that we use as the basis of our case study is a networked logging service. As shown in Figure 1, this service consists of client applications that generate log records and send them to a central logging server that receives and stores the log records for later inspection and processing.

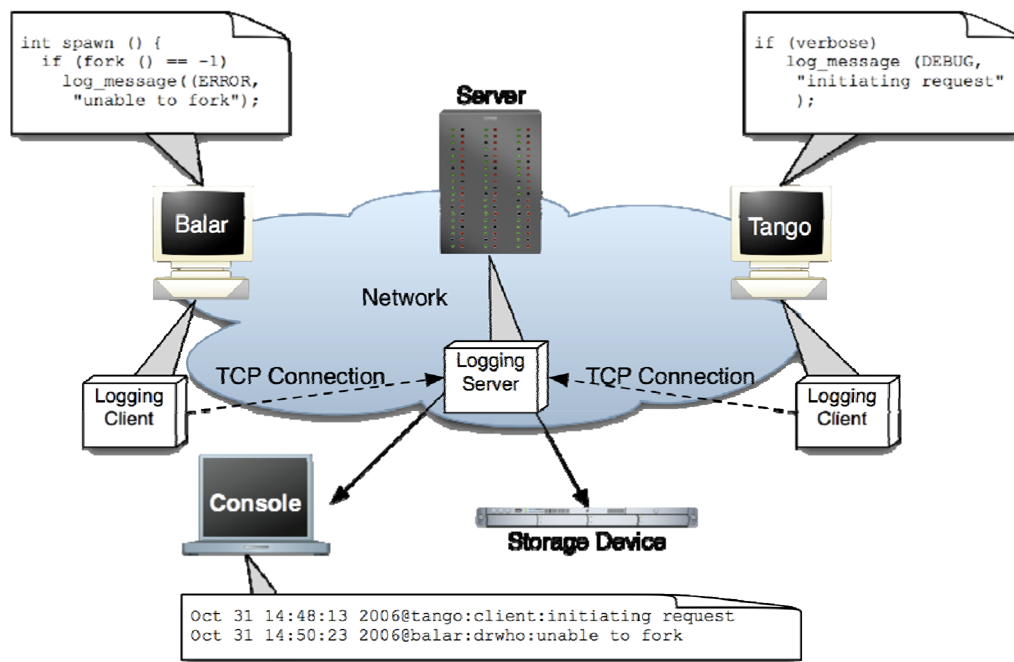

Figure 1. Architecture of a networked logging service

The logging server portion of our networked logging service provides an ideal context for demonstrating the beauty of OO networked software because it exhibits the following dimensions of design-time variability that developers can choose from when implementing such a server:

- Different interprocess communication (IPC) mechanisms (such as sockets, SSL, shared memory, TLI, named pipes, etc.) that developers can use to send and receive log records.
- Different concurrency models (such as iterative, reactive, thread-per-connection, process-per-connection, various types of thread pools, etc.) that developers can use to process log records.
- Different locking strategies (such as thread-level or process-level recursive mutex, non-recursive mutex, readers/writer lock, null mutex, etc.) that developers can use to serialize access to resources, such as a count of the number of requests, shared by multiple threads.
- Different log record formats can be transmitted from client to server, and once received by the server the log records can be handled in different ways, e.g., printed to console, stored to a single file, or even one file per client to maximize parallel writes to disk.

It is relatively straightforward to implement any one of these combinations, such as socket-based IPC, thread-per-connection concurrency model, and a thread-level non-recursive mutex logging server. A one-size-fits-all solution, however, is inadequate to meet the needs of all logging services because different customer requirements and different operating environments can significantly impact time/space tradeoffs, cost, and schedule. A key challenge is therefore to design a configurable logging server that is *easily extensible* to meet new needs with a *minimum of effort*. At the heart of the solution to this challenge is a thorough understanding of the patterns and associated design techniques needed to develop OO frameworks that efficiently

- Capture common structure and behavior in base classes and generic classes
- Enable selective customization of behavior via subclasses and by providing concrete parameters to generic classes.

Figure 2 illustrates the design of an OO logging server framework that realizes the approach outlined above. The core of this design is the `Logging_Server` class, which defines the common structure and functionality for the logging server via the use of

- C++ parameterized types, which allow developers to defer the selection of data types used in generic classes or functions until their point of instantiation.
- The Template Method pattern [GoF], which defines the skeleton of an algorithm, delegating individual steps to methods which may be overridden by subclasses
- The Wrapper Façade pattern [POSA2], which encapsulates non object-oriented APIs and data within type-safe object-oriented classes.

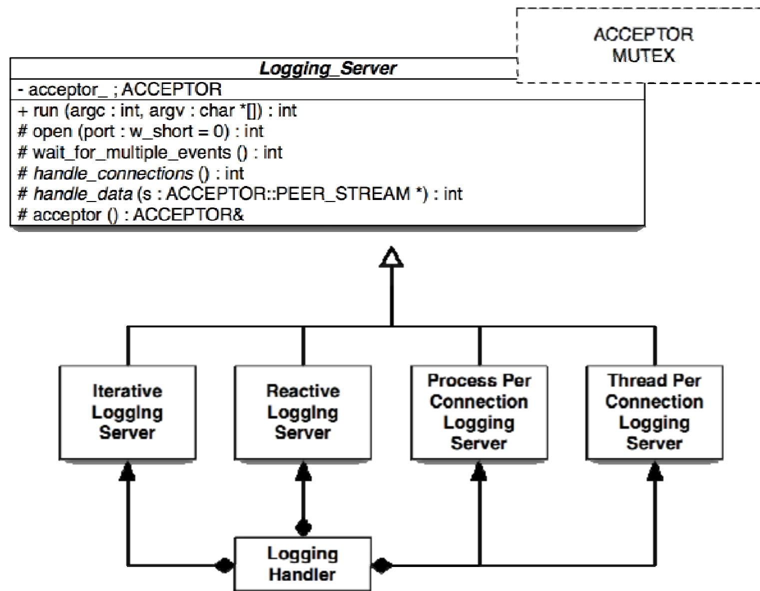

Figure 2. Object-oriented design for the logging server framework

Subclasses and concrete instantiations of `Logging_Server` refine this common reusable architecture to customize variable steps in the logging server behavior by selecting desired IPC mechanisms, concurrency models, and locking strategies. The `Logging_Server` is thus a product-line architecture [PLA] that defines an integrated set of classes that collaborate to define a reusable design for a family of related logging servers.

The remainder of this chapter is organized as follows. The next section describes the OO design of the logging server framework, exploring the architecture of the logging server framework and the forces that influence the design of the OO framework to motivate why we selected certain patterns and language features, as well as summarize alternative approaches that rejected for various reasons. Two further sections present several C++ sequential programming instantiations of the logging server framework and concurrent programming instantiations of this framework. We conclude by summarizing the beauty of the OO software concepts and techniques in this chapter.

Object-Oriented Design of the Logging Server Framework

Before we discuss the OO design of our logging server, it is important to understand several key concepts about OO frameworks. Most programmers are familiar with the concept of a class library, which is a set of reusable classes that provides functionality that may be used when developing their OO programs. OO frameworks extend the benefits of OO class libraries in the following ways [Johnson]:

- **They define "semi-complete" applications that embody domain-specific object structures and functionality**. Classes in a framework work together to provide a generic architectural skeleton for applications in a particular domain, such as graphical user interfaces, avionics mission computing, or networked logging services. Complete applications can be composed by inheriting from and/or instantiating framework components. In contrast, class libraries are less domain-specific and provide a smaller

scope of reuse. For instance, class library components like classes for strings, complex numbers, arrays, and bitsets are relatively low-level and ubiquitous across many application domains.

- **Frameworks are active and exhibit "inversion of control" at run-time**. Class libraries are typically passive, i.e., they perform isolated bits of processing when invoked by threads of control from self-directed application objects. In contrast, frameworks are active, i.e., they direct the flow of control within an application via event dispatching patterns, such as Reactor [POSA2] and Observer [GoF]. The ``inversion of control'' in the run-time architecture of a framework is often referred to as "The Hollywood Principle," which states "Don't call us, we'll call you" [Hollywood].

Frameworks are typically designed by analyzing various potential problems that the framework might address and identifying which parts of each solution are the same and which areas of each solution are unique. This design method is called *commonality/variability analysis* [CVA], which covers the following topics: (1) *scope*, which defines the domains (i.e., problem areas a framework addresses) and context of the framework, (2) *commonalities*, which describe the attributes that recur across all members of the family of products based on the framework, and (3) *variabilities*, which describe the attributes unique to the different members of the family of products.

Understanding the Commonalities

The first step in designing our logging server framework is therefore to understand the parts of the system that should be implemented by the framework (commonalities) and parts of the system left to be specialized in subclasses or parameters (variabilities). This analysis is straightforward because the steps involved in processing a log record sent over a network can be decomposed into the steps shown in Figure 3, which are common to all logging server implementations.

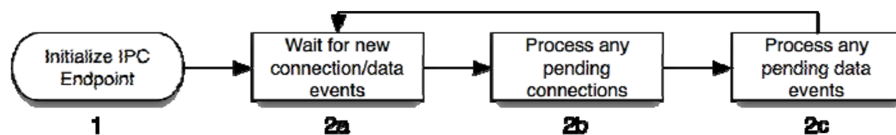

Figure 3. Logging server main loop

During this stage of the design process we define each step as abstractly as possible. For example, at this stage we've made minimal assumptions about the type of IPC mechanisms, other than they are connection-oriented to ensure reliable delivery of log records. Likewise, we've avoided specifying the type of concurrency strategy (e.g., whether the server can handle multiple requests, and if so how they are dispatched) or the synchronization mechanism used by each step. The actual choice of specific behavior for a step is thus deferred to the subsequent concrete implementation(s) that provide a particular variant for each step.

The Template Method pattern [GoF] is a useful way to define abstract steps and defer implementation of their specific behavior to later steps in the design process. This pattern defines a base class that implements the common—but abstract—steps in the *template method* in terms of *hook methods* that can be overridden selectively by concrete implementations. Programming language features, such as pure virtual functions in C++ or ab-

stract methods in Java, can be used to ensure that all concrete implementations define the hook methods. Figure 4 shows the structure of the Template Method pattern and demonstrates how this pattern is applied to the design of our OO logging server framework.

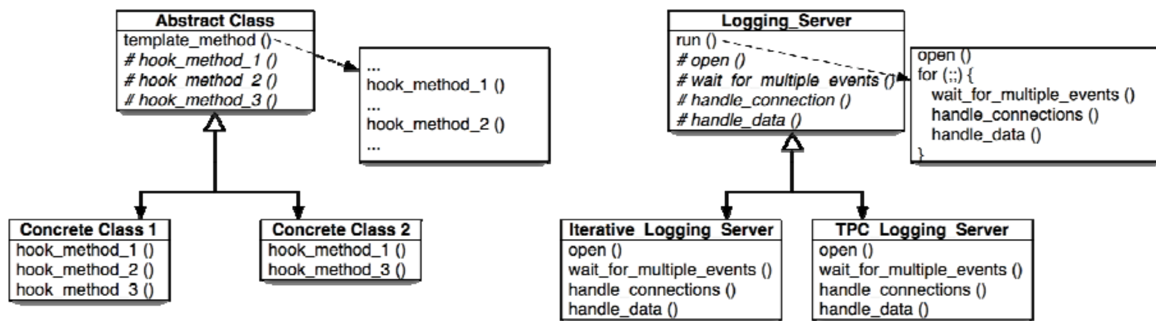

Figure 4. Template method pattern and applying template method to the OO logging server framework

Accommodating Variation

Although the Template Method pattern addresses the overall design of the steps in our logging server framework, we're left with the question of how to accommodate all three dimensions of variability defined earlier (i.e., IPC, concurrency, and synchronization mechanisms) needed to support our design. One approach would simply use the Template Method pattern and implement one IPC/concurrency/synchronization combination per concrete subclass. Unfortunately, this approach would yield exponential growth in the number of concrete subclasses, as each addition to any dimension could generate another implementation for each possible combination of the other dimensions. A pure Template Method design, therefore, would not be substantially better than handcrafting one-off implementations of a logging server for each variant.

A more effective and scalable design could leverage the fact that our variability dimensions are largely independent. The choice of a different IPC, for instance, is unlikely to require changes in the concurrency or synchronization mechanisms used. Moreover, there is a high-level commonality in how different types of IPC and synchronization mechanisms function, e.g., IPC mechanisms can initiate/accept connections and send/receive data on connections, whereas synchronization mechanisms have operations to acquire and release locks. The design challenge is to encapsulate the accidental complexities in these APIs so that they can be used interchangeably.

A solution to this challenge is to use the Wrapper Façade pattern [POSA2], which presents a single unified OO interface for the underlying non-OO IPC and synchronization mechanisms provided by system functions in an OS. Wrapper facades are particularly useful for enhancing portability by hiding accidental complexities between mechanisms, as well as making it less tedious and error-prone to work with these APIs. For instance, a wrapper façade can define a higher-level type system that ensures only correct operations are called on the underlying non-OO (and less type-safe) OS IPC and synchronization data structures. The role of a wrapper façade is shown in Figure 5.

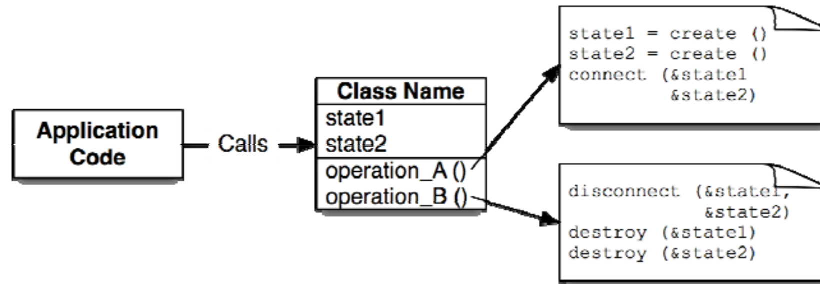

Figure 5. Wrapper façade design pattern

ACE [C++NPv1] is widely used example of host infrastructure middleware that defines unified OO interfaces using wrapper façades for both IPC and synchronization mechanisms. We base the wrappers façades in this chapter on simplified versions of those provided by ACE. Figure 6 shows some of the wrapper facades provided by ACE.

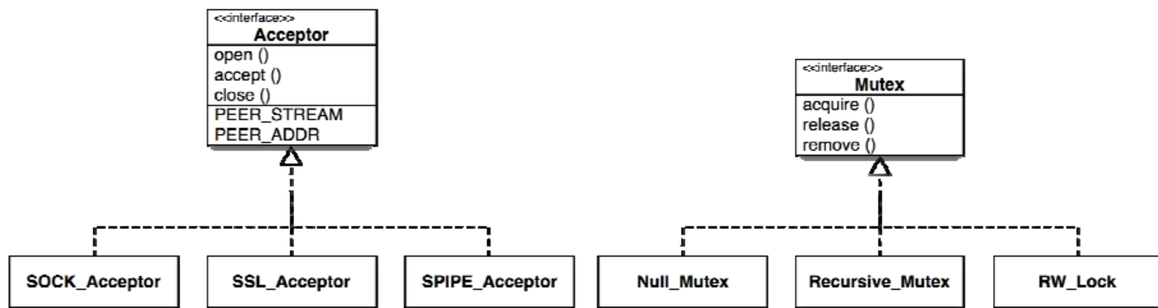

Figure 6. Some ACE wrapper facades for passive connection establishment and synchronization

The `Acceptor` wrapper façade provides the means to create passive mode connections and provides "traits" to represent aspects of a mechanism that work essentially the same way across different implementations, just with different APIs. For instance, `PEER_STREAM` and `PEER_ADDR`, designate dependant wrapper facades appropriate for sending/receiving data and for addressing by the IPC mechanism, respectively. `SOCK_Acceptor` is a subclass of `Acceptor` used in this chapter to implement a factory for passively establishing connections implemented using the socket API.

The `Mutex` wrapper façade provides an interface whose methods acquire and release locks, including a `Recursive_Mutex` implemented using a mutex that will not deadlock when acquired multiple times by the same thread a `RW_Lock` that implements readers/writer semantics, and a `Null_Mutex` whose `acquire()/release()` methods are inline no-ops. The latter class is an example of the Null Object pattern [Woolf] and is useful for eliminating synchronization without changing application code. Figure 6 makes it appear as if each family of classes is related by inheritance, but they are actually implemented by classes unrelated by inheritance that have a common interface and can be used as type parameters to C++ templates. We made this design choice to avoid virtual method call overhead.

Tying it All Together

Another design challenge is how to associate a concurrency strategy with an IPC and synchronization mechanism. One approach would be to use the Strategy pattern [GoF], which encapsulates algorithms as objects so they can be swapped at run-time. This approach would provide the `Logging_Server` with a pointer to abstract base classes of `Acceptor` and `Mutex` and then rely on dynamic binding and polymorphism to dispatch the virtual methods to the appropriate subclass instances.

While a Strategy-based approach is feasible, it is not ideal. Each incoming log record may generate several calls to methods in the `Acceptor` and `Mutex` wrapper facades. Performance could therefore degrade since virtual methods incur more overhead than non-virtual method calls. Given that dynamically swapping IPC or synchronization mechanisms is not a requirement for our logging servers, a more efficient solution is to use C++ parameterized types to instantiate our logging server classes with the wrapper façades for IPC and synchronization. We therefore define the following generic abstract base class called `Logging_Server` from which all logging servers in this chapter will inherit:

```
template <typename ACCEPTOR, typename MUTEX>
class Logging_Server {
 public:
   typedef Log_Handler<typename ACCEPTOR::PEER_STREAM> HANDLER;

   Logging_Server (int argc, const char *argv);

   // Template method that runs each step in the main event loop.
   virtual void run (void);

protected:
   // Hook methods that enable each step to be varied.
   virtual void open (void);
   virtual void wait_for_multiple_events (void) = 0;
   virtual void handle_connections (void) = 0;
   virtual void handle_data
                (typename ACCEPTOR::PEER_STREAM *stream = 0) = 0;

   // Increment the request count, protected by the mutex.
   virtual void count_request (size_t number = 1);

   // Instance of template parameter that accepts connections.
   ACCEPTOR acceptor_;

   // Keeps a count of the number of log records received.
   size_t request_count_;

   // Instance of template parameter that serializes access to
   // the request_count_.
   MUTEX mutex_;

   // Address that the server will listen on for connections.
   std:string server_address_;
};
```

Most methods in `Logging_Server` are pure virtual, which ensures that subclasses implement them. The `open()` and `count_request()` methods shown below, however, are reused by all logging servers in this chapter:

```
template <typename ACCEPTOR, typename MUTEX>
Logging_Server<ACCEPTOR, MUTEX>::Logging_Server
   (int argc, char *argv[]): request_count_ (0) {
  // Parse the argv arguments and store the server address_...
}

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::open (void) {
  return acceptor_.open (server_address_);
}

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::count_request (size_t number) {
  mutex_.acquire (); request_count_ += number; mutex_.release ();
}
```

The `Log_Handler` class is responsible for demarshaling a log record from a connected data stream whose IPC mechanism is designated by the `ACCEPTOR` type parameter. The implementation of this class is outside the scope of this chapter, and could itself be another dimension of variability, e.g., logging servers might want to support different log message formats. If we were to support varying the format of method of storing incoming log messages, this class could be yet another template parameter in our logging framework. For our purposes, it is sufficient to know that it is parameterized by the IPC mechanism and provides two methods: `peer()`, which returns a reference to the data stream, and `log_record()`, which reads a single log record from the stream.

The primary entry point into `Logging_Server` is the template method called `run()`, which implements the steps outlined in Figure 3, delegating the specific steps to the hook methods declared in the protected section of `Logging_Server`, as shown in the code fragment below:

```
template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::run (void) {
  try {
    // Step 1: initialize an IPC factory endpoint to listen for
    // new connections on the server address.
    open ();

    // Step 2: Go into an event loop
    for (;;) {
      // Step 2a: wait for new connections or log records
      // to arrive.
      wait_for_multiple_events ();

      // Step 2b: accept a new connection (if available)
      handle_connections ();

      // Step 2c: process received log record (if available)
      handle_data ();
    }
  } catch (...) { /* ... Handle the exception ... */ }
}
```

The beauty of this code is that

- Its pattern-based design makes it easy to handle variation in concurrency models, e.g., by varying the behavior of the `run()` template method by providing specific implementation of the hook methods in the implementation of subclasses and
- Its template-based design makes it easy to handle variation in IPC and synchronization mechanisms, e.g., by plugging in different types into the `ACCEPTOR` and `MUTEX` template parameters.

Implementing Sequential Logging Servers

This section demonstrates the implementation of logging servers that feature sequential concurrency models, i.e., all processing is performed in a single thread. We cover both iterative and reactive implementations of sequential logging servers below.

An Iterative Logging Server

Iterative servers process all log records from each client before handling any log records from the next client. Since there is no need to spawn or synchronize threads we use the `Null_Mutex` façade to parameterize the `Iterative_Logging_Server` subclass template, as follows:

```
template <typename ACCEPTOR>
class Iterative_Logging_Server :
  virtual Logging_Server<ACCEPTOR, Null_Mutex> {
public:
  typedef Logging_Server<ACCEPTOR, Null_Mutex>::HANDLER HANDLER;
  Iterative_Logging_Server (int argc, char *argv[]);

protected:
  virtual void open (void);
  virtual void wait_for_multiple_events (void) {};
  virtual void handle_connections (void);
  virtual void handle_data
                   (typename ACCEPTOR::PEER_STREAM *stream = 0);
  HANDLER log_handler_;

  // One log file shared by all clients.
  std::ofstream logfile_;
};
```

Implementing this version of our server is straightforward. The `open()` method decorates the behavior of the method from the `Logging_Server` base class by opening an output file before delegating to the parent's `open()`, as follows:

```
template <typename ACCEPTOR> void
Interative_Logging_Server<ACCEPTOR>::open (void) {
  logfile_.open (filename_.c_str ());
  if (!logfile_.good ()) throw std::runtime_error;
  // Delegate to the parent's open() method.
  Logging_Server<ACCEPTOR, Null_Mutex>::open ();
}
```

The `wait_for_multiple_events()` method is a no-op. It is not needed because we just handle a single connection at any one time. The `handle_connections()` method therefore simply blocks until a new connection is established, as follows:

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_connections (void)
{ acceptor_.accept (log_handler_.peer ()); }
```

Finally, `handle_data()` simply reads log records from the client and writes them to the log file until the client closes the connection or an error occurs:

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_data (void) {
    while (log_handler_.log_record (logfile _))
      count_request ();
}
```

While the iterative server is straightforward to implement, it suffers from the drawback of only being able to service only a one client at a time. A second client that attempts to connect may time out while waiting for the first to finish its request.

A Reactive Logging Server

The reactive logging server alleviates one of the primary drawback with the iterative logging server described above by processing multiple client connections and log record requests via operating system synchronous event demultiplexing APIs provided by the OS, such as `select()` and `WaitForMultipleObjects()`. These APIs can monitor multiple clients by waiting in a single thread of control for I/O-related events to occur on a group of I/O handles and then interleave the processing of log records. Since a reactive logging server is still fundamentally sequential, however, it inherits from the iterative logging server implemented earlier, as shown in Figure 7.

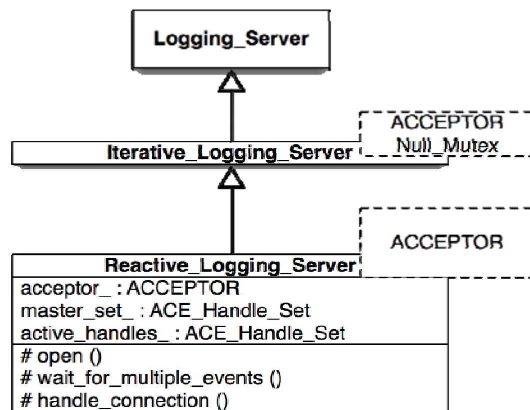

Figure 7. Reactive logging server interface

The `Reactive_Logging_Server` class overrides all four hook methods that it inherits from base class `Iterative_Logging_Server`. Its `open()` hook method decorates the behavior of the base class method to initialize the `ACE_Handle_Set` member variables, which are part of the wrapper façades that simplify the use of `select()`, as shown below:

```
template <typename ACCEPTOR> void
Reactive_Logging_Server<ACCEPTOR>::open () {
  // Delegate to base class.
  Iterative_Logging_Server<ACCEPTOR>::open ();

  // Mark the handle associated with the acceptor as active.
  master_set_.set_bit (acceptor_.get_handle ());

  // Set the acceptor's handle into non-blocking mode.
  acceptor_.enable (NONBLOCK);
}
```

The `wait_for_multiple_events()` method is needed in this implementation, unlike its counterpart in `Iterative_Server`. As shown in Figure 8, this method uses a synchronous event demultiplexer (in this case, `select()`) to detect which I/O handles have connection or data activity pending.

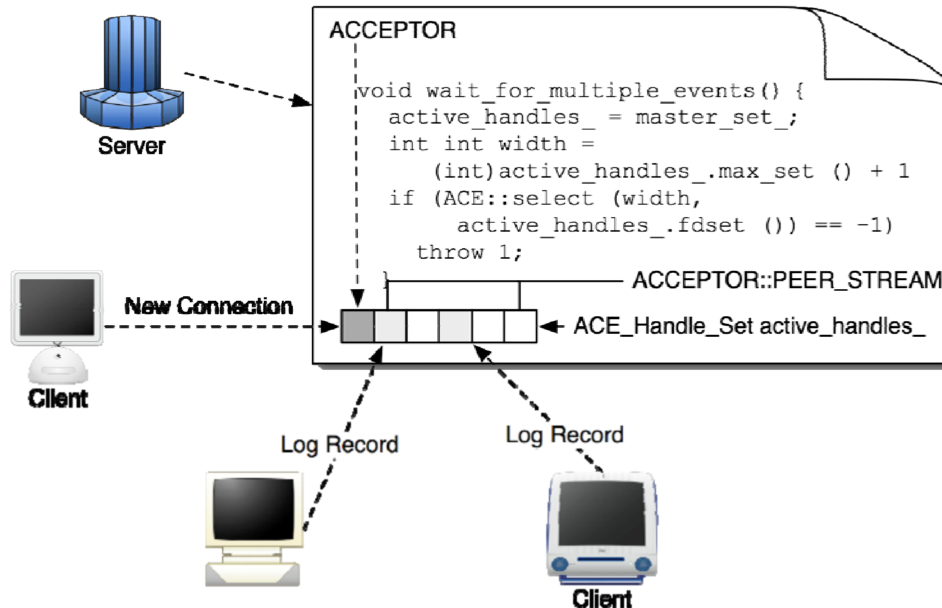

Figure 8. Using a synchronous event demultiplexer in the `Reactive_Logging_Server`

After `wait_for_multiple_events()` has executed, the `Reactive_Logging_Server` has a cached a set of handles with pending activity (i.e., either new connection requests or new incoming data events), which will then be handled by its other two hook methods: `handle_connections()` and `handle_data()`. The `handle_connections()` method checks whether the acceptors handle is active, and if so, accepts as many connections as possible and caches them in the `master_handle_set_`. Similarly, the `handle_data()` method iterates over the remaining active handles marked by `select()` earlier. This activity is simplified by the ACE socket wrapper façade that implements an instance of the Iterator pattern [GoF] for socket handle sets, as shown in Figure 9.

The following code implements a `Reactive_Logging_Server` main program that uses the socket API:

```
int main (int argc, char *argv[]) {
  Reactive_Logging_Server<SOCK_Acceptor> server (argc, argv);
  server.run ();
  return 0;
}
```

The first line of our main function parameterizes the `Reactive_Logging_Server` with the `SOCK_Acceptor` type, which will cause the C++ compiler to generate code for a reactive logging server that is able to communicate over sockets. This will, in turn, parameterize its `Logging_Server` base class with both the `SOCK_Acceptor` and `Null Mutex`, by virtue of the hard-coded template argument provided when we inherited from it. The second line calls the run () template method, which is delegated to the `Logging_Server`

base class, which itself delegate to the various hook methods we implemented in this class.

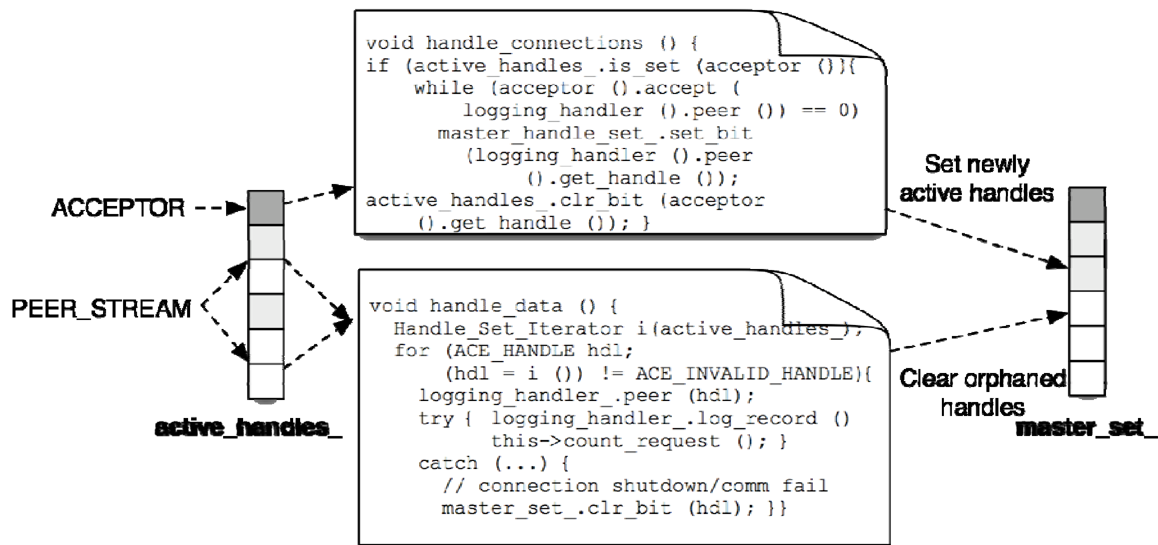

Figure 9. Reactive server connection/data event handling

Evaluating the Sequential Logging Server Solutions

The `Reactive_Logging_Server` improves upon the `Iterative_Logging_Server` by interleaving its servicing of multiple clients, rather than just handling one client in its entirety at a time. It does not take advantage of OS concurrency mechanisms, however, so it cannot leverage multi-processors effectively to process multiple log records in parallel. It also cannot overlap computation and communication by processing log records while reading new records. These limitations impede its scalability as the number of clients increases, even if the underlying hardware supports multiple simultaneous threads of execution.

Although the `Iterative_Logging_Server` and `Reactive_Logging_Server` only run in a single thread of control—and are thus not scalable for most production systems—their simplicity highlights several more beautiful aspects of our OO framework-based design:

- Our use of hook methods in the `Logging_Server::run()` template method shielded application developers from low-level details, e.g., how a logging server performs IPC and event demulxiplexing operations, thereby enabling the developers to focus on domain-specific application logic by leveraging the expertise of framework designers.
- Our use of wrapper façades in allows us to lock/unlock mutexes, listen on a particular IPC mechanism to accept new connections, and wait for multiple I/O events concisely, efficiently, and portably. Without these useful abstractions, we would have had to write many lines of tedious and error-prone code that would be hard to understand, debug, and evolve.

The benefits from these abstractions become more apparent with more complex concurrent logging servers shown below, as well as with more complex framework use cases, such as graphical user interfaces [GoF] or communication middleware [POSA2].

Implementing Concurrent Logging Servers

To overcome the scalability limitations of the iterative and reactive servers shown in Section 3, the logging servers in this section use both OS concurrency mechanisms: processes and threads. Using the APIs provided by operating systems to spawn threads or processes, however, can be a daunting task due to accidental complexities in their design. These complexities stem from semantic and syntactic differences that exist not only between different operating systems, but also different versions of the same operating system. Our solution to these complexities is again to apply wrapper façades that provide a consistent interface across platforms and integrate these wrapper facades into our OO `Logging_Server` framework.

A Thread-per-Connection Logging Server

Our thread-per-connection logging server (`TPC_Logging_Server`) uses a main thread that waits for and accepts new connections from clients. After accepting a new connection, a new worker thread will be spawned to handle incoming log records from that connection. Figure 10 shows the steps in this process.

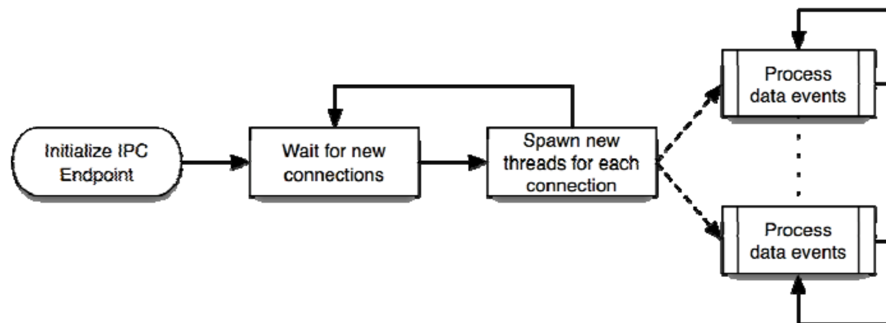

Figure 10. Steps in the thread-per-connection logging server

The main loop for this particular logging server differs from the steps depicted in Figure 3 since the call to `handle_data()` is not necessary, as the worker threads are responsible for that call. There are two ways to handle this situation:

1. We could note that the base `run()` method calls `handle_data()` with the default argument of a `NULL` pointer, and simply have our implementation exit immediately for that input.

2. We could simply override the `run()` method with our own implementation that omits this call.

Solution two may at first appear advantageous since it avoids a virtual method call to `handle_data()`. Solution one is better in this case, however, since the performance hit of that virtual call is not a limiting factor and overriding the `run()` template method would prevent this class from benefiting from changes to the base class implementation, potentially causing it to fail in subtle and pernicious ways.

The main challenge here is implementing the concurrency strategy itself. As with the `Iterative_Server` in Section 3.1, the `wait_for_multiple_events()` method is superfluous because our main loop simply waits for new connections, so it is sufficient for `handle_connections()`to block on `accept()` and subsequently spawn worker threads to handle connected clients. Our `TPC_Logging_Server` class must therefore provide a method to serve as an entry point for the thread. In C and C++, a class method may only serve as an entry point to a thread if it is defined as static, so we define the `TPC_Logging_Server::svc()`static class method.

At this point, we have an important design decision to make: what exactly does the thread entry point do? It is tempting to simply have the `svc()` method itself perform all of the work necessary to receive log records from its associated connection. This design is less than ideal, however, since static methods cannot be virtual, which will cause problems if we later derive a new logging server from this implementation to change the way it handles data events. Application developers would then be forced to provide an implementation of `handle_connections()` that is textually identical to this class to call the proper static method. Moreover, to leverage our existing design and code, it is preferable to have the log record processing logic inside the `handle_data()` method and define a `Thread_Args` helper object that holds the peer returned from `accept()` and a pointer to the `Logging_Server` object itself. Our class interface will therefore look like the diagram in Figure 11.

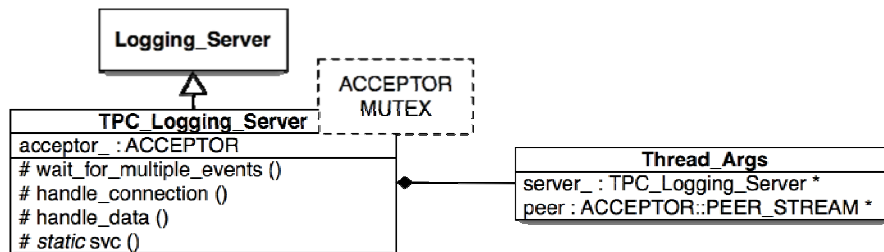

Figure 11. Thread-per-connection server interface

The remainder of `TPC_Logging_Server` is straightforward to implement, requiring only that our thread entry point delegate processing to the virtual method `handle_data()` using the `server_pointer` contained within the `Thread_Args` helper object passed to the `svc()` method, as shown in Figure 12.

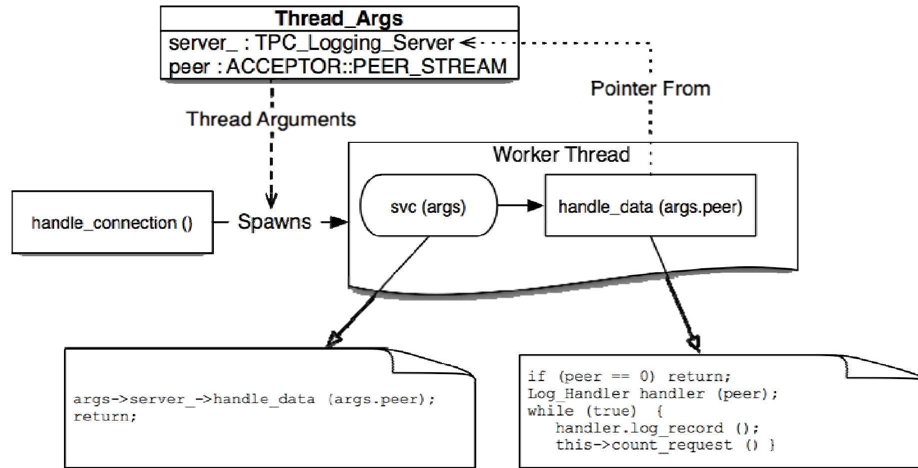

Figure 12. Thread-per-connection thread behavior

The following code implements a `TPC_Logging_Server` main program that uses the secure socket API and the readers/writer lock:

```
int main (int argc, char *argv[]) {
  TPC_Logging_Server<SSL_Acceptor, RW_Lock> server (argc, argv);
  server.run ();
  return 0;
}
```

This `main()` function instantiates a `TPC_Logging_Server` that communicates using SSL connections and uses a `RW_Lock` to synchronize the `count_connections()` function in the `Logging_Server` base class. Except for the name of the class we are instantiating, this `main()` function is identicalto the one that was written earlier in this chapter for the `Reactive_Logging_Server`. This commonality is another beautiful aspect of our design: regardless of the particular combination of concurrency, IPC, and synchronization mechanisms we choose to use, the instantiation and invocation of our server remains the same.

The thread-per-connection logging service addresses the scalability limitations with the sequential implementations described in Section 3. The design of our OO framework made it straightforward to integrate this concurrency model with minimal changes to the existing code. In particular, `TPC_Logging_Server` inherits implementations of `open()`, `count_request()`, and most importantly `run()`, allowing this class to leverage bugfixes and improvements to our main event loop transparently. Moreover, adding the necessary synchronization around the `request_count_` is simply a matter of parameterizing the `TPC_Logging_Server` with the `RW_LOCK` class.

A Process-per-Connection Logging Server

The process-per-connection logging server described next is similar to the thread-per-connection design shown in Figure 10, except that instead of spawning a thread, we spawn a new process to handle incoming log records from each client. The choice of processes over threads for concurrency forces us to make design choices to accommodate the variations in process creation semantics between platforms. There are two key se-

mantic differences between the process APIs on Linux and Windows that our server design must encapsulate

- In Linux (and other POSIX systems) the primary vehicle for creating new processes is the `fork()` system function, which generates an exact duplicate of the calling program image, including open I/O handles, differing only in their return value from `fork()`. At this point, child processes can choose to proceed from that point, or load a different program image using the `exec*()` family of system calls.
- Windows, however, uses the `CreateProcess()` API call, which is functionally equivalent to a POSIX `fork()` followed immediately by a call to one of the `exec*()` system functions. The impact of this difference is that in Windows you have an *entirely new process* that by default *does not* have access to I/O handles open in the parent. To use a connection accepted by the parent process, therefore, the handle must be explicitly duplicated and passed to the child on the command line.

We therefore define a set of wrapper façades that not only hide the syntactic differences between platforms, but also provide a way to hide the semantic differences as well. These wrappers consist of the three cooperating classes shown in Figure 13. The `Process` class represents a single process and is used to create and synchronize processes. The `Process_Options` class provides a way to set both platform-independent (such as command line options and environment variables) and platform-specific process options (such as avoiding zombie processes). Finally, the `Process_Manager` class portably manages the lifecycle of groups of processes. We won't cover all the use of these wrapper façades in this chapter, though they are based on the wrapper facades in ACE [C++NPv1]. It is sufficient to know that not only can processes be created portably on Linux and Windows, but also that I/O handles can be duplicated and passed portably and automatically to the new process.

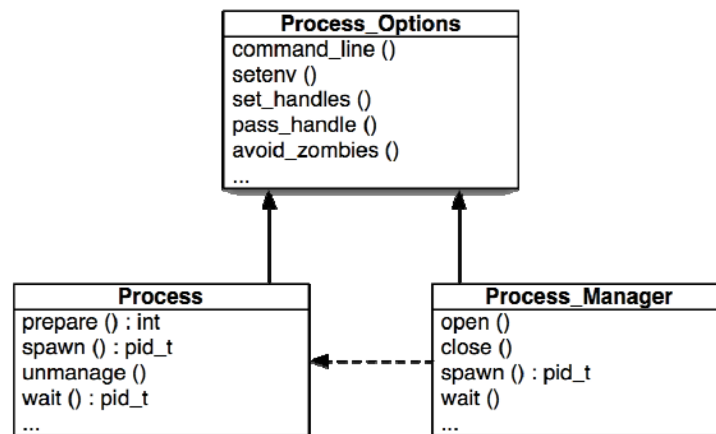

Figure 13. Portable process wrapper façades

The design challenge is therefore to accommodate the fact that processes spawned after new connections are accepted will start at the beginning of our program. We certainly don't want child processes to attempt to open a new acceptor and listen for connections of their own—instead, they should only listen for data events on their assigned handle. A

naive solution to this problem would rely on applications to detect this condition and call a special entry point defined in the interface to our process based `Logging_Server` class.

This simple solution, however, is less than ideal since not only would it require us to change the public interface of our process-based `Logging_Server`, but it would also expose intimate implementation details to applications, violating encapsulation. A better solution is to override the `run()` template method inherited from the `Logging_Server` base class, which is passed a copy of the command-line argument by users, to determine if it has been passed any I/O handles. If not, the process assumes it is a parent and delegates to the base class `run()` method. Otherwise, the process assumes it's a child so it decodes the handle and calls `handle_data()`, as shown in Figure 14.

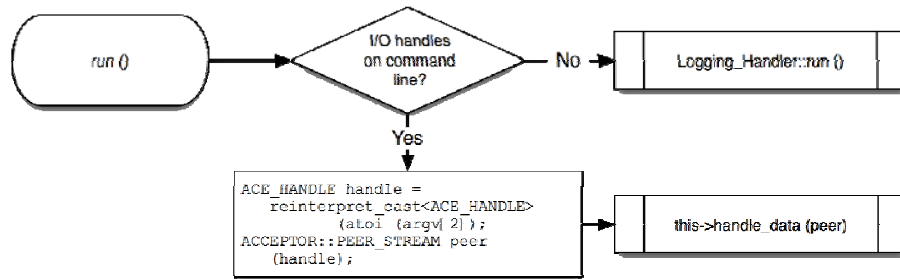

Figure 14. Process-per-connection `run()` template method

The remainder of this server implementation is straightforward. As shown in Figure 15, the process wrapper façade makes the procedure for spawning our worker processes fairly simple. The implementation for `handle_data()` should be textually identical to that shown in Figure 12.

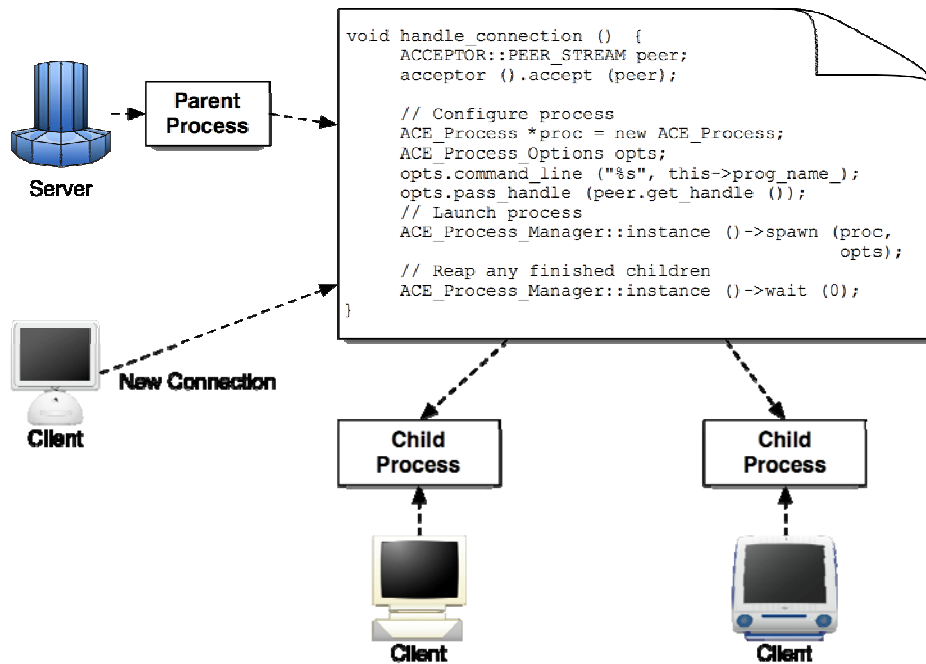

Figure 15. Connection handling for the process-per-connection server

Our reimplementation of the `run()` method from the `Logging_Server` base class allows us to maintain the beautifully simple, straightforward, and uniform invocation used by our other logging servers:.

```
int main (int argc, char *argv[]) {
  PPC_Logging_Server<SSL_Acceptor, Null_Mutex> server (argc, argv);
  server.run ();
  return 0;
}
```

This `main()` program differs from the thread-per-connection server only in the name of the class that is instantiated, and the fact that we have chosen to use a `Null_Mutex` to provide synchronization. The dispatch of either a parent or a child process is handled transparently by the `run()` method, driven by the command-line arguments passed to the `PPC_Logging_Server` constructor.

Evaluating the Concurrent Logging Server Solutions

Both concurrent logging servers described in this section significantly enhance the `Reactive_Logging_Server` and `Iterative_Logging_Server` in terms of their ability to scale as the number of clients increases by taking leveraging hardware and OS support for multiple threads of execution. It is hard, however, to develop thread-per-connection and process-per-connection concurrency strategies in a platform-agnostic manner. We accomplished this task by using wrapper façades to hide platform differences. Our framework-based server design also provided a common external interface to the `Logging_Server` class, shielding the bulk of the logging server from the configured concurrency strategy. Moreover, our design leveraged the the `run()` template method inherited from the `Logging_Server` base class, allowing our implementations to integrate bugfixes or other enhancements to the main server event loop.

Concluding Remarks

The logging server application presented in this chapter provides a digestible—yet realistic—vehicle for showing how to apply OO design/programming techniques, patterns, and frameworks to implementation software for networked applications. In particular, our OO framework demonstrates a number of beautiful design elements, ranging from its abstract design to concrete elements in the implementations of the different concurrency models. Our design also used C++ features, such as templates and virtual functions, in conjunction with design patterns, such as Wrapper Façade and the Template Method, to create a family of logging servers that is portable, reusable, flexible, and extensible.

The Template Method pattern in the `Logging_Server` base class's `run()` method allowed us to define common steps in a logging server, deferring specialization of individual steps of its operation to hook methods in derived classes. While this pattern helped factor out common into the base class, it did not adequately address all our required points of variability, such as synchronization and IPC mechanisms, For these remaining dimensions, therefore, we used the Wrapper Façade pattern to hide semantic and syntactic differences, ultimately making use of these dimensions entirely orthogonal to the implementation of individual concurrency models. This design allowed us to use parameterized classes to address these dimensions of variability, which increased the flexibility of our framework without affecting its performance adversely.

Finally, our individual implementations of concurrency models, such as thread-per-connection and process-per-connection, used wrapper façades to make their implementations more elegant and portable. The end result is a labor-saving software architecture that enables developers to reuse common design and programming artifacts, as well as provide a means to encapsulate variabilities in a common, parameterizeable way.

A concrete implementation of this logging server framework may be found on the web at www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/ACE/examples/Beautiful_Code and in-the ACE distribution in ACE_wrappers/examples/Beautiful_Code.

References

C++NPv1 D. C. Schmidt, S. D. Huston: C++ Network Programming – Volume 1, Addison-Wesley, 2002

C++NPv2 D. C. Schmidt, S. D. Huston: C++ Network Programming – Volume 2, Addison-Wesley, 2003

CVA J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering" *IEEE Software*, 15(6) November/December, 37—45, 1998

GoF E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Hollywood John Vlissides, "Pattern Hatching - Protection, Part I: The Hollywood Principle," C++ Report, February 1996.

JNP E. Harold, Java Network Programming, Second Edition
Second Edition, O'Reilly, 2000

Johnson R. Johnson, Frameworks = Patterns + Components, Communications of the ACM, Volume 40, Number 10, October 1997.

OOSC B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall 1997.

PLA P. Clements and L. Northrop, Software Product-lines: Practices and Patterns, Addison-Wesley, ISBN 0201703327, August 20, 2001

POSA2 D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann: Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects, John Wiley and Sons, 2000

Woolf B. Woolf, "The Null Object Pattern," Pattern Languages of Program Design Volume 3, Addison-Wesley, 1997.