Design Patterns and Frameworks for Concurrent CORBA Event Channels

Douglas C. Schmidt
Washington University, St. Louis

http://www.cs.wustl.edu/~schmidt/
schmidt@cs.wustl.edu
Motivation

- Asynchronous messaging and group communication are important for real-time applications.

- This example explores the design patterns and reusable framework components used in an OO architecture for CORBA Real-time Event Channels.

- CORBA Event Channels route events from Supplier(s) to Consumer(s).
Communication Models for Event Channels

(A) THE CANONICAL PUSH MODEL
(B) THE CANONICAL PULL MODEL

(C) THE HYBRID PUSH/PULL MODEL
(D) THE HYBRID PULL/PUSH MODEL
OO Software Architecture of the
Event Channel

- Consumer Proxy
- Consumer Proxy
- Consumer Proxy
- Reactor
- Supplier Proxy
- Supplier Proxy
- QoS Queues
- Event Channel
The Event Channel components are based upon a system of design patterns.
Design Patterns in the Event Channel (cont’d)

- **Reactor**
  - “Decouples event demultiplexing and event handler dispatching from application services performed in response to events”

- **Half-Sync/Half-Async**
  - “Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency”

- **Active Object**
  - “Decouples method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads”
Using the Reactor Pattern for the Single-Threaded Event Channel
Event Channel Inheritance

Hierarchy

- Task
- Proxy Handler
- Supplier Proxy
- Consumer Proxy
- Message Queue

APPLICATION-INDEPENDENT

APPLICATION-SPECIFIC
IO_Proxy Class Public Interface

• Common methods and data for I/O Proxys

    // Keeps track of events sent and received.
typedef u_long COUNTER;

    // This is the type of the Consumer_Map.
typedef Null_Mutex MAP_LOCK;
typedef Map_Manager <Event_Header,
                 Consumer_Set,
                 MAP_LOCK> CONSUMER_MAP;

class Proxy_Handler : public Task<Null_Synch>
{
public:
    // Initialize the Proxy.
    virtual int open (void * = 0);

private:
    static COUNTER events_sent_; 
    static COUNTER events_received_;
Supplier_Proxy Interface

- Handle input processing and routing of events from Suppliers

```cpp
class Supplier_Proxy : public Proxy_Handler {
    protected:
        // Notified by Reactor when Supplier event arrives.
        virtual int handle_input (void);

        // Low-level method that receives an event from a Supplier.
        virtual int recv (Message_Block *&);

        // Forward an event from a Supplier to Consumer(s).
        int forward (Message_Block *);
};
```
Consumer_Proxy Interface

- Handle output processing of events sent to Consumers

```cpp
class Consumer_Proxy : public Proxy_Handler
{
public:
    // Send an event to a Consumer.
    virtual int push (Message_Block *);

protected:
    // Perform a non-blocking push() (will
    // may queue if flow control occurs).
    int nonblk_push (Message_Block *event);

    // Finish sending an event when flow control
    // abates.
    virtual int handle_output (void);

    // Low-level method that sends an event to
    // a Consumer.
    virtual int send (Message_Block *);
};
```
Collaboration in Single-threaded Event Channel Forwarding

1: handle_input()
2: recv(event)
3: find()
4: push (event)
5: send(event)
6: push (event)
7: send(event)
8: enqueue(event)
9: schedule_wakeup()
10: handle_output()
11: dequeue(event)
12: send(event)
// Receive input event from Supplier and forward // the event to Consumer(s).

int
Supplier_Proxy::handle_input (void)
{
    Message_Block *event = 0;

    // Try to get the next event from the // Supplier.
    if (recv (event) == COMPLETE_EVENT)
    {
        Proxy_Handler::events_received_++;
        forward (event);
    }
}

// Send an event to a Consumer (queue if necessary).

int
Consumer_Proxy::push (Message_Block *event)
{
    if (msg_queue ()->is_empty ())
        // Try to send the Message_Block *without* blocking!
        nonblk_put (event);
    else
        // Events are queued due to flow control.
        msg_queue ()->enqueue_tail (event);
}
// Forward event from Supplier to Consumer(s).

int
Supplier_Proxy::forward (Message_Block *event)
{
    Consumer_Set *c_set = 0;

    // Determine route.
    Consumer_Map::instance ()->find (event, c_set);

    // Initialize iterator over Consumers(s).
    Set_Iterator<Consumer_Proxy *> iter (c_set);

    // Multicast event.
    for (Consumer_Proxy *ch;
         si.next (ch) != -1;
         si.advance ()) {
        // Make a "logical copy" (via reference counting).
        Message_Block *new_event = event->duplicate ();

        if (ch->push (new_event) == -1) // Drop event.
            new_event->release (); // Decrement reference count.
    }

    event->release (); // Delete event.
}
Event Structure

- An Event contains two portions
  - The Event_Header identifies the Event
    - Used for various types of filtering

```cpp
and correlation
class Event_Header {
public:
    Supplier_Id s_id_;  
    int priority_;        
    EventType type_;     
    time_t time_stamp_;   
    size_t length_;       
};
```

- The Event contains a header plus a variable-sized message

```cpp
class Event {
public:
    // The maximum size of an event.
    enum { MAX_PAYLOAD_SIZE = /* ... */ };  
    Event_Header header_; // Fixed-sized header portion.  
    char payload_[MAX_PAYLOAD_SIZE]; // Event payload.  
};
```
Q: What should happen if `push()` fails?

- e.g., if a Consumer queue becomes full?

A: The answer depends on whether the error handling policy is different for each router object or the same...

- Bridge/Strategy pattern: give reasonable default, but allow substitution

A related design issue deals with avoiding output blocking if a Consumer connection flow controls
Q: How can a flow controlled ConsumerProxy know when to proceed again without polling or blocking?

A: Use the Event_Handler::handle_output notification scheme of the Reactor

- i.e., via the Reactor’s methods schedule_wakeup and cancel_wakeup

This provides cooperative multi-tasking within a single thread of control

- The Reactor calls back to the handle_output method when the ConsumerProxy is able to transmit again
Performing Non-blocking Push Operations

- The following method will push the event without blocking

  - We need to queue if flow control conditions occur

```c
int Consumer_Proxy::nonblk_push (Message_Block *event)
{
  // Try to send the event using non-blocking I/O
  if (send (event) == EWOULDBLOCK)
  {
    // Queue in *front* of the list to preserve order.
    msg_queue ()->enqueue_head (event);

    // Tell Reactor to call us when we can send again.
    Service_Config::reactor ()->schedule_wakeup
      (this, Event_Handler::WRITE_MASK);
  }
  else
    Proxy_Handler::events_sent_++;
}
```
/ * Finish sending an event when flow control
  * conditions abate. This method is automatically
  * called by the Reactor.

int
Consumer_Proxy::handle_output (void)
{
    Message_Block *event = 0;

    // Take the first event off the queue.
    msg_queue ()->dequeue_head (event);

    if (nonblk_push (event) != 0)
    {
        // If we succeed in writing msg out completely
        // (and as a result there are no more msgs
        // on the Message_Queue), then tell the Reactor
        // not to notify us anymore.

        if (msg_queue ()->is_empty ()
            Service_Config::reactor ()->cancel_wakeup
            (this, Event_Handler::WRITE_MASK);
    }
}
Event_Channel Class Public Interface

- Maintains maps of the Consumer_Proxy object references and the Supplier_Proxy object references

// Parameterized by the type of I/O Proxys.
template<class Supplier_Proxy, // Supplier policies
class Consumer_Proxy> // Consumer policies
class Event_Channel
{
    public:
        // Perform initialization.
        virtual int init (int argc, char *argv[]);

        // Perform termination.
        virtual int fini (void);

    private:
        // ...
};
Dynamically Configuring Services into an Application

- Main program is generic

```c
// Example of the Service Configurator pattern.

int main (int argc, char *argv[]) {
    Service_Config daemon;
    // Initialize the daemon and
    // dynamically configure services.
    daemon.open (argc, argv);

    // Run forever, performing configured services.
    daemon.run_reactor_event_loop ();

    /* NOTREACHED */
}
```
Dynamic Linking an Event_Channel Service

- Service configuration file

```bash
$ cat ./svc.conf
static Svc_Manager "-p 5150"
dynamic Event_Channel_Service Service_Object *
    Event_Channel.dll:make_Event_Channel () "-d"
```

- Application-specific factory function used to dynamically link a service

```c
// Dynamically linked factory function that allocates
// a new single-threaded Event_Channel object.

extern "C" Service_Object *make_Event_Channel (void);

Service_Object *
make_Event_Channel (void)
{
    return new Event_Channel<Supplier_Proxy,
                        Consumer_Proxy>;
    // ACE automatically deletes memory.
}
```
Concurrent Strategies for Event Channel

- The single-threaded Event Channel has several limitations
  1. Fragile program structure due to cooperative multi-tasking
  2. Doesn’t take advantage of multi-processing platforms

- Therefore, a concurrent solution may be beneficial
  - Though it can also increase concurrency control overhead

- The following slides illustrate how OO techniques push this decision to the “edges” of the design
  - This greatly increases reuse, flexibility, and performance tuning
Using the Active Object Pattern for the Multi-threaded Event Channel

- Registered Objects
  - FRAMEWORK
    - Timer Queue
    - Handle Table
    - Signal Handlers
  - APPLICATION
    - Consumer Proxy: Event Handler
    - Consumer Proxy: Event Handler
  - KERNEL
    - Supplier Proxy: Event Handler
    - Event Handler

OS EVENT DEMULTIPLEXING INTERFACE
Collaboration in the Active Object-based Event Channel Forwarding
Half-Sync/Half-Async Pattern

• **Intent**
  
  – “Decouple synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency”

• This pattern resolves the following forces for concurrent communication systems:

  – *How to simplify programming for higher-level communication tasks*
    
    ▶ These are performed synchronously (via Active Objects)

  – *How to ensure efficient lower-level I/O communication tasks*
    
    ▶ These are performed asynchronously (via Reactor)
Structure of the Half-Sync/Half-Async Pattern

- **SYNC TASK 1**
- **SYNC TASK 2**
- **SYNC TASK 3**

1, 4: read(data)

3: enqueue(data)

2: interrupt

MESSAGE QUEUES

ASYNC TASK

EXTERNAL EVENT SOURCES
Using the Half-Sync/Half-Async Pattern in the Event_Channel

1: dequeue(event)
2: send(event)
3: forward(event)
4: enqueue(event)

1: dispatch()
2: recv(event)
3: forward(event)
4: enqueue(event)

MESSAGE QUEUES

: Consumer Proxy
: Consumer Proxy
: Consumer Proxy

: Supplier Proxy
: Supplier Proxy

: Reactor

SYNCHRONOUS TASK LAYER
ASYNCHRONOUS TASK LAYER
QUEUEING LAYER
Configuring Synchronization Mechanisms

// Determine the type of synchronization mechanism.
#if defined (ACE_USE_MT)
typedef MT_SYNCH SYNCH;
#else
    typedef NULL_SYNCH SYNCH;
#endif /* ACE_USE_MT */

typedef Null_Mutex MAP_LOCK;

// This is the type of the Consumer_Map.
typedef Map_Manager <Event_Header,
          Consumer_Set,
          MAP_LOCK> CONSUMER_MAP;

class Proxy_Handler : public Task<SYNCH>
{ /* ... */ };
• **Q:** What is the `MT_SYNCH` class and how does it work?

• **A:** `MT_SYNCH` provides a thread-safe synchronization policy for a particular instantiation of a `Svc_Handler`

  – e.g., it ensures that any use of a `Svc_Handler`'s `Message_Queue` will be thread-safe

  – Any Task that accesses shared state can use the “traits” in the `MT_SYNCH`

  ```
  class MT_SYNCH { public:
    typedef Mutex MUTEX;
    typedef Condition<Mutex> CONDITION;
  };
  ```

  – Contrast with `NULL_SYNCH`

  ```
  class NULL_SYNCH { public:
    typedef Null_Mutex MUTEX;
    typedef NullCondition<Null_Mutex> CONDITION;
  };
  ```
Thr_Consumer_Proxy Class Interface

- New subclass of Proxy_Handler uses the Active Object pattern for the Consumer_Proxy
  - Uses multi-threading and synchronous I/O to transmit events to Consumers
  - Transparently improve performance on a multiprocessor platform and simplify design

#define ACE_USE_MT
#include "Proxy_Handler.h"

class Thr_Consumer_Proxy : public Proxy_Handler
{
public:
    // Initialize the object and spawn a new thread.
    virtual int open (void *);

    // Send an event to a Consumer.
    virtual int push (Message_Block *);

private:
    // Transmit Supplier events to Consumer within separate thread.
    virtual int svc (void);
Thr_Consumer_Proxy Class
Implementation

- The multi-threaded version of open is slightly different since it spawns a new thread to become an active object!

```c
// Override definition in the Consumer_Proxy class.

int
Thr_Consumer_Proxy::open (void *)
{
    // Become an active object by spawning a
    // new thread to transmit events to Consumers.

    activate (THR_NEW_LWP | THR_DETACHED);
}
```

- activate is a pre-defined method on class Task
// Queue up an event for transmission (must not block
// since all Supplier_Proxys are single-threaded).

int
Thr_Consumer.Proxy::push (Message_Block *event)
{
    // Perform non-blocking enqueue.
    msg_queue ()->enqueue_tail (event);
}

// Transmit events to the Consumer (note simplification
// resulting from threads...)

int
Thr_Consumer.Proxy::svc (void)
{
    Message_Block *event = 0;
    // Since this method runs in its own thread it
    // is OK to block on output.

    while (msg_queue ()->dequeue_head (event) != -1) {
        send (event);
        Proxy_Handler::events_sent_++;
    }
}
Dynamic Linking an Event_Channel Service

- Service configuration file

```bash
% cat ./svc.conf
remove Event_Channel_Service
dynamic Event_Channel_Service Service_Object *
    thr_Event_Channel.dll:make_Event_Channel () "-d"
```

- Application-specific factory function used to dynamically link a service

```c
// Dynamically linked factory function that allocates
// a new multi-threaded Event_Channel object.

extern "C" Service_Object *make_Event_Channel (void);

Service_Object *
make_Event_Channel (void)
{
    return new Event_Channel<Supplier_Proxy,
                         Thr_Consumer_Proxy>;
    // ACE automatically deletes memory.
}
```
Eliminating Race Conditions

• **Problem**
  
  – The concurrent Event Channel contains “race conditions” *e.g.*, 

    ▶ Auto-increment of static variable `events_sent` is not serialized properly

• **Forces**
  
  – Modern shared memory multi-processors use *deep caches* and *weakly ordered* memory models

  – Access to shared data must be protected from corruption

• **Solution**
  
  – Use synchronization mechanisms
Basic Synchronization
Mechanisms

- One approach to solve the serialization problem is to use OS mutual exclusion mechanisms explicitly, e.g.,

```c
// SunOS 5.x, implicitly "unlocked".
mutex_t lock;

int
Thr_Consumer_Proxy::svc (void)
{
    Message_Block *event = 0;
    // Since this method runs in its own thread it
    // is OK to block on output.

    while (msg_queue ()->dequeue_head (event) != -1) {
        send (event);
        mutex_lock (&lock);
        Proxy_Handler::events_sent_++;
        mutex_unlock (&lock);
    }
}
```
Problems Galore!

- Adding these `mutex_*` calls explicitly is *inelegant, obtrusive, error-prone, and non-portable*
  - *Inelegant*
    - “Impedance mismatch” with C/C++
  - *Obtrusive*
    - Must find and lock all uses of `events_sent_`
  - *Error-prone*
    - C++ exception handling and multiple method exit points cause subtle problems
      - Global mutexes may not be initialized correctly...
  - *Non-portable*
    - Hard-coded to Solaris 2.x
C++ Wrappers for Synchronization

- To address portability problems, define a C++ wrapper:

```cpp
class Thread_Mutex
{
public:
    Thread_Mutex (void) {
        mutex_init (&lock_, USYNCH_THREAD, 0);
    }

    ~Thread_Mutex (void) { mutex_destroy (&lock_-); }

    int acquire (void) { return mutex_lock (&lock_-); }
    int tryacquire (void) { return mutex_trylock (&lock_-); }
    int release (void) { return mutex_unlock (&lock_-); }

private:
    mutex_t lock_; // SunOS 5.x serialization mechanism.

    void operator= (const Thread_Mutex &);
    Thread_Mutex (const Thread_Mutex &);
};
```

- Note, this mutual exclusion class interface is portable to other OS platforms
Porting Thread_Mutex to Windows NT

- Win32 version of Thread_Mutex

```cpp
class Thread_Mutex
{
public:
    Thread_Mutex (void) {
        InitializeCriticalSection (&lock_);
    }
    ~Thread_Mutex (void) {
        DeleteCriticalSection (&lock_);
    }
    int acquire (void) {
        EnterCriticalSection (&lock_); return 0;
    }
    int tryacquire (void) {
        TryEnterCriticalSection (&lock_); return 0;
    }
    int release (void) {
        LeaveCriticalSection (&lock_); return 0;
    }
private:
    CRITICAL_SECTION lock_; // Win32 locking mechanism. // ...
Using the C++ Thread_Mutex Wrapper

- Using the C++ wrapper helps improve portability and elegance:

```c
Thread_Mutex lock;

int Thr_Consumer_Proxy::svc (void)
{
    Message_Block *event = 0;

    while (msg_queue ()->dequeue_head (event) != -1) {
        send (event);
        lock.acquire ();
        Proxy_Handler::events_sent_++;
        lock.release ();
    }
}
```

- However, it does not solve the *obtrusiveness* or *error-proneness* problems...
Automated Mutex Acquisition and Release

- To ensure mutexes are locked and unlocked, we’ll define a template class that acquires and releases a mutex automatically

```cpp
template <class LOCK>
class Guard
{
public:
    Guard (LOCK &m): lock_ (m) { lock_.acquire (); }  
    ~Guard (void) { lock_.release (); } 
    // ...
private:
    LOCK &lock_; 
}
```

- Guard uses the C++ idiom whereby a constructor acquires a resource and the destructor releases the resource
Using the Guard Class

- Using the Guard class helps reduce errors:

```cpp
Thread_Mutex lock;

int Thr_Consumer_Proxy::svc (void)
{
    Message_Block *event = 0;
    // Since this method runs in its own thread it
    // is OK to block on output.

    while (msg_queue ()->dequeue_head (event) != -1) {
        send (event);
        {
            // Constructor releases lock.
            Guard<Thread_Mutex> mon (lock);
            Proxy_Handler::events_sent_++;
            // Destructor releases lock.
        }
    }
}
```

- However, using the Thread_Mutex and Guard classes is still overly obtrusive and subtle (may lock too much scope...)

42
Q: Why is Guard parameterized by the type of LOCK?

A: there are many locking mechanisms that benefit from Guard functionality, e.g.,

- Non-recursive vs recursive mutexes
- Intra-process vs inter-process mutexes
- Readers/writer mutexes
- Solaris and System V semaphores
- File locks
- Null mutex

In ACE, all synchronization classes use the Wrapper and Adapter patterns to provide identical interfaces that facilitate parameterization.
The Wrapper Pattern

- **Intent**
  
  - “Encapsulate low-level, stand-alone functions within type-safe, modular, and portable class interfaces”

- This pattern resolves the following forces that arises when using native C-level OS APIs

  1. *How to avoid tedious, error-prone, and non-portable programming of low-level IPC and locking mechanisms*

  2. *How to combine multiple related, but independent, functions into a single cohesive abstraction*
Structure of the Wrapper Pattern

1: request ()

2: specific_request()
Using the Wrapper Pattern for Locking

1: acquire()

client

Mutex
acquire()
release()
tryacquire()

Solaris
mutex_lock()
mutex_unlock()
mutex_trylock()

2: mutex_lock()
Using the Adapter Pattern for Locking

1: Guard()

2: acquire()

3: mutex_lock()
The following C++ template class uses the “Decorator” pattern to define a set of atomic operations on a type parameter:

template <class LOCK = Thread_Mutex, class TYPE = u_long>
class Atomic_Op {
public:
    Atomic_Op (TYPE c = 0) { count_ = c; }

    TYPE operator++ (void) {
        Guard<LOCK> m (lock_); return ++count_;
    }

    operator TYPE () {
        Guard<LOCK> m (lock_);
        return count_;
    }
    // Other arithmetic operations omitted...

private:
    LOCK lock_;  
    TYPE count_;  
};
Using Atomic.Op

- A few minor changes are made to the class header:

  ```c
  #if defined (MT_SAFE)
  typedef Atomic.Op<> COUNTER; // Note default parameters...
  #else
  typedef Atomic.Op<ACE_Null_Mutex> COUNTER;
  #endif /* MT_SAFE */
  ```

- In addition, we add a lock, producing:

  ```c
  class Proxy_Handler
  {
  // ...

    // Maintain count of events sent.
    static COUNTER events_sent_;
  }
  ```
Thread-safe Version of Consumer_Proxy

- `events_sent_` is now serialized automatically and we only lock the minimal scope necessary.

```c
int Thr_Consumer_Proxy::svc (void)
{
    Message_Block *event = 0;

    // Since this method runs in its own thread it
    // is OK to block on output.

    while (msg_queue ()->dequeue_head (event) != -1) {
        send (event);
        // Calls Atomic_Op<>::operator++.
        Proxy_Handler::events_sent_++;
    }
}
```
Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures
- Patterns explicitly capture expert knowledge and design tradeoffs
- Patterns help improve developer communication
- Patterns help ease the transition to object-oriented technology
Drawbacks to Design Patterns

- Patterns do not lead to direct code reuse

- Patterns are deceptively simple

- Teams may suffer from pattern overload

- Patterns are validated by experience rather than by testing

- Integrating patterns into a software development process is a human-intensive activity
Suggestions for Using Patterns Effectively

- **Do not recast everything as a pattern**
  - Instead, develop strategic domain patterns and reuse existing tactical patterns

- **Institutionalize rewards for developing patterns**

- **Directly involve pattern authors with application developers and domain experts**

- **Clearly document when patterns apply and do not apply**

- **Manage expectations carefully**
Patterns Literature

• Books
  – Gamma et al., “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley, 1994
  – *Pattern Languages of Program Design* series by Addison-Wesley, 1995 and 1996

• Special Issues in Journals
  – October ’96 “Communications of the ACM” (guest editors: Douglas C. Schmidt, Ralph Johnson, and Mohamed Fayad)

• Magazines
  – C++ Report and Journal of Object-Oriented Programming, columns by Coplien, Vlissides, and Martin
Obtaining ACE

- The ADAPTIVE Communication Environment (ACE) is an OO toolkit designed according to key network programming patterns

- All source code for ACE is freely available
  - Anonymously ftp to wuarchive.wustl.edu
  - Transfer the files /languages/c++/ACE/*\.gz

- Mailing lists
  * ace-users@cs.wustl.edu
  * ace-users-request@cs.wustl.edu
  * ace-announce@cs.wustl.edu
  * ace-announce-request@cs.wustl.edu

- WWW URL