CSC9006 Real-Time Systems

Pattern-Based C++ Network Programming
Part 2: Design Dimensions and Patterns

Chao Wang

Networked Cyber-Physical Systems Laboratory
Department of Computer Science and Information Engineering
National Taiwan Normal University

June 27th, 2021

NATIONAL TAIWAN NORMAL UNIVERSITY

Agenda

Link to the video recording for my presentation: (click here).

Complexities in Networked Application Development

- It takes nontrivial effort to achieve timely and dependable performance
- Inherent complexities:
    - Suitable choices of communication protocols and mechanisms
    - Effective uses of concurrency designs
    - Sensible knobs for service configuration
- Accidental complexities:
    - Lacks of type safety and portability
    - Re-discoveries and re-inventions of core networked application concepts and capabilities

Exercise:

Think of those programs you've ever developed. Is it inherent complexities, accidental complexities, or both, that made the development challenging?

Design Dimensions for Networked Applications

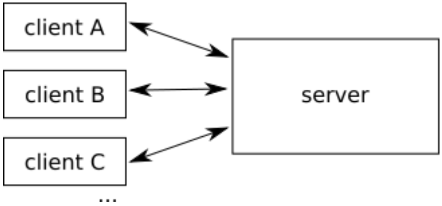
- Domain analysis and its benefits:
  - providing a common vocabulary of domain abstractions
  - enhancing some reuse of mature, time-tested designs
- Design dimensions to manage the inherent complexities:
  - Communication dimensions (e.g., rules and forms)
  - Concurrency dimensions (e.g., policies and mechanisms for the use of processes and threads)
  - Service dimensions (i.e., key properties of the service itself)
  - Configuration dimensions (e.g., how services are identified and bound)

Exercise:

Do some research to answer why typically a networked service would separate ‘client connection’ from ‘client request invocation.’

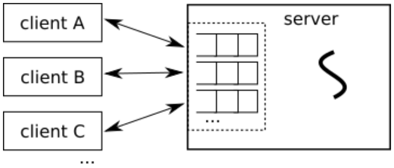
The Concurrency Design Dimension

- Types of server designs:
  - iterative
  - reactive
  - concurrent
- Thread Spawning Strategies
- Patterns for concurrency design



Iterative Server

- The server handles each client request in its entirety before servicing the next request.

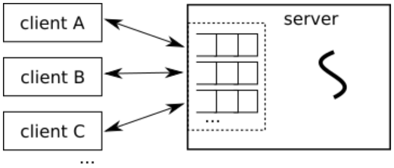


Example pseudo-code

```
void iterative_server () {
    for (each client request) {
      perform requested service;
}}
```

Reactive Server

- Essentially an iterative server; in addition, requests from multiple endpoints were handled in round-robin order.

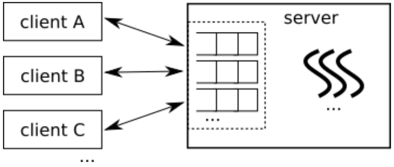


Example pseudo-code

```
void reactive_server () {
    for (;;) {  // event loop
        select() on multiple endpoints;
        for (each active endpoint) {
            perform the requested service;
}}}
```

Concurrent Server

- The server uses multiple threads to handle requests simultaneously.



Example pseudo-code

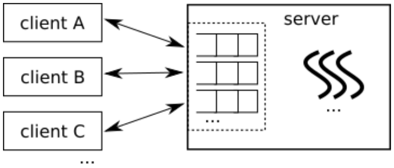
```
void master_thread () {
    for (each client request) {
        spawn a new worker thread and
         pass the request to the thread;}}
void worker_thread () {
    perform the requested service;
    terminate itself;}
```

Questions to Ponder

- Iterative server vs. reactive server: why would developers ever choose to use an iterative server?
- Reactive server vs. concurrent server: for Internet-of-Things applications with low data rates, would you choose a reactive server or a concurrent server?
- Concurrent server: rather than spawning a new thread per request, what about using some pre-spawned threads and reusing them?
- If a client asked for a response from the server, how would this further complicate the server design?

Patterns for Concurrency Design

- Background: threading, and data passing between threads
- One worker thread
  - Pattern 1: Active Object
- Multiple worker threads (i.e., a thread pool)
  - Pattern 2: Half-Sync/Half-Async
  - Pattern 3: Leader/Followers



Threading, and Data Passing Between Threads

- Observations from multi-threaded programming:
  - Many common uses of threading follow a certain pattern
  - A queue is often needed when passing data between threads
  - Thread synchronization primitives need very-careful placements
- Two of the design principles in ACE:
  - Simplify for the common case
  - Use hierarchies to enhance design clarity and extensibility

The ACE Way of Threading

- The ACE_Task_Base class captures the common use pattern of threading:

```
class Our_filter : public ACE_Task_Base {
public:
    virtual int svc (void) {...}
...
}

int ACE_TMAIN (int, ACE_TCHAR * []) {
    Our_filter filter;
    int result = filter.activate ();
    filter.wait ();
    ...
}
```

The ACE Way of Threading (cont.)

- The ACE_Guard set of classes would acquire the specified lock when they are constructed, and would release the lock when they are destructed.
- The ACE_NEW_RETURN macro handles both object creating and error checking.

```
int Our_filter::update_table (int event_id) {
    ACE_Guard<ACE_Thread_Mutex> guard (this->mutex_);
    ...
    ACE_NEW_RETURN (object, Object, -1);
    ...
    // Guard is destoryed upon return.
}
```

The ACE Way of Data Passing

- The ACE_Task class (derived from ACE_Task_Base) has a built-in queue. A thread may pass some data to another thread by pushing the data to the built-in queue of the target thread.
- An example implementation for the master thread (Slide 8):

```
class Master : public ACE_Task<ACE_MT_SYNCH>
{ // ...
    virtual int svc (void) {
        Worker worker_thread;
        worker_thread.activate ();
        // ...and after we got a client request
        ACE_Message_Block *data = NULL;
        worker_thread.putq (data);
        ...
}
```

The ACE Way of Data Passing (cont.)

- `ACE_Task` also help synchronizes the accesses for the built-in queue.
- An example implementation of the worker thread (Slide 8):

```
class Worker : public ACE_Task<ACE_MT_SYNCH>
{
    ...
    virtual int svc (void) {
        while (1) {
            ACE_Message_Block *data = NULL;
            this->getq (data);
            process_data (data);
        }
        ...
}
```

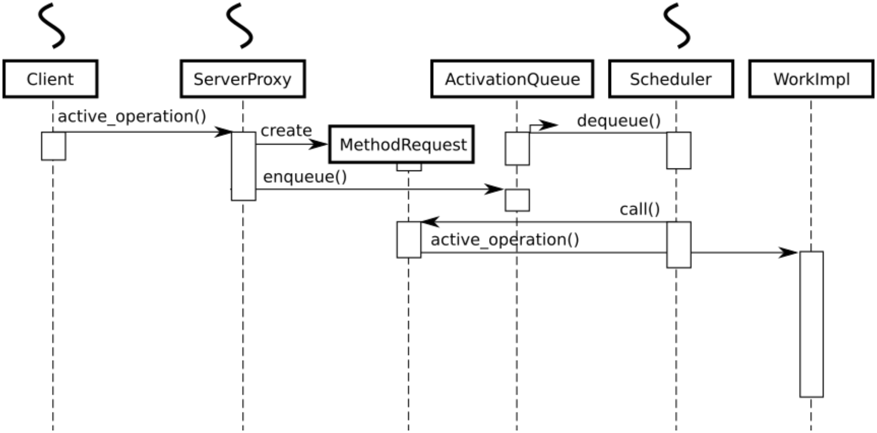
The Active Object Pattern

- Motivation: we want to have two threads communicate via what look like regular method calls; the methods should be executed in the context of the receiver.

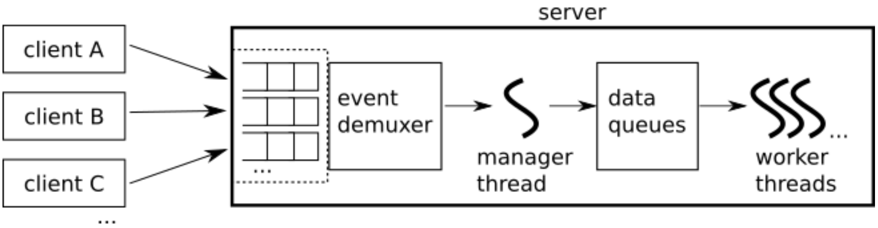
Example C++ Code

```
int main () {
  ActiveWorker guy; // an ACE_Task_Base object
  guy.activate();
  ...
  guy.work(); // we want this to occur in
              // the context of the guy thread,
              // i.e., this is an asynchronous call.
  ...
```

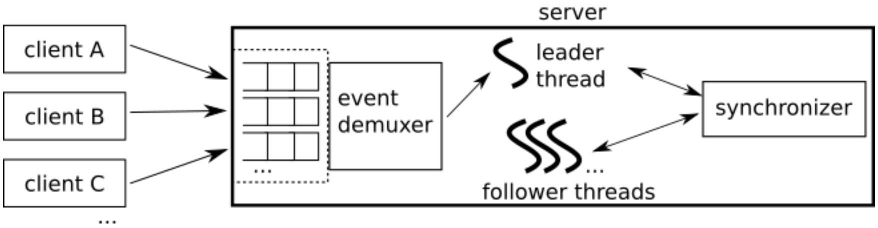
Sequence Diagram of The Active Object Pattern



The Half-Sync/Half-Async Pattern



The Leader/Followers Pattern



References

[1] Schmidt, Douglas, and Stephen D. Huston. C++ Network Programming, Volume I: Mastering Complexity with ACE and Patterns. FT Press, 2001. (Chapters 0 and 5)

[2] Huston, Stephen D., James CE Johnson, and Umar Syyid. The ACE programmer's guide: practical design patterns for network and systems programming. Addison-Wesley Professional, 2004. (Chapters 12, 15, and 16)