Network Applications: Async Servers and Operational Analysis

Y. Richard Yang
http://zoo.cs.yale.edu/classes/cs433/
10/03/2013

Recap: Async Network Server
- Basic idea: non-blocking operations
  - asynchronous initiation (e.g., aio_read) and completion notification (callback)
  - peek system state to issue only ready operations

Recap: Java Async I/O Basis: OS State Polling
- Example: a system call to check if sockets are ready for ops.

Recap: Basic Dispatcher Structure
// clients register interests/handlers on events/sources
while (true) {
  - ready events = select() /* or selectNow(), or select(int timeout) */
    is call to check the ready events from the registered interest events of sources */

  - foreach ready event {
    switch event type:
      - accept: call accept handler
      - readable: call read handler
      - writable: call write handler
  }
}

Recap: Java Dispatcher v1
while (true) {
  - selector.select()
  - Set readyKeys = selector.selectedKeys();

  - foreach key in readyKeys {
    switch event type of key:
      - accept: call accept handler
      - readable: call read handler
      - writable: call write handler
  }
}

See AsyncEchoServer/v1/EchoServer.java

Admin
- Assignment Three will be posted later today.
- Dates for the two exams
Recap: Two Problems of V1

- Empty write
- Still read after no longer having data to read

Finite State Machine (FSM)

- Finite state machine after fixing the two issues
- Problem of the finite state machine?

Fix Remaining Empty Write

- Why no need to introduce FSM for a thread version?

Another State Machine

Comparing FSMs

- V2:
  - Mixed read and write
- Example last slide: staged
  - First read request and then write response
- Exact design depends on application, e.g.,
  - HTTP/1.0 channel may use staged
  - Chat channel may use mixed
Extending v2

Many real programs run the dispatcher in a separate thread to allow main thread to interact with users
-> start dispatcher in its own thread

Protocol specific coding, not reusable
-> derive an async/io TCP server software framework so that porting it to a new protocol involves small edits (e.g., defining read/write handlers)

Extensible Dispatcher Design

Attachment stores generic event handler
Define interfaces
- IAcceptHandler and
- IReadWriteHandler
Retrieve handlers at run time

if (key.isReadable() || key.isWritable()) {
    IReadWriteHandler rwH = (IReadWriteHandler) key.attachment();
    if (key.isReadable()) rwH.handleRead(key);
    if (key.isWritable()) rwH.handleWrite(key);
}

Dispatcher Interface

Register a channel to be selected and its handler object
Update interest of a selectable channel
Deregister

Handler Design: Acceptor

What should an accept handler object know?
- ServerSocketChannel (so that it can call accept)
  - Can be derived from SelectionKey in the call back
- Dispatcher (so that it can register new connections)
  - Need to be passed in constructor or call back
What ReadWrite object to create (different protocols may use different ones)?
- Pass a Factory object: SocketReadWriteHandlerFactory

Handler Design: ReadWriteHandler

What should a ReadWrite handler object know?
- SocketChannel (so that it can read/write data)
  - Can be derived from SelectionKey in the call back
- Dispatcher (so that it can change state)
  - Need to be passed in constructor or in call back
Class Diagram of v3

- Dispatcher
  - registerNewSelection()
  - deregisterSelection()
  - updateInterests()

- ChannelHandler
  - handleException()

- AcceptHandler
  - handleAccept()

- IChannelHandler
  - handleException()

- IAcceptHandler
  - handleAccept()

- IReadWriteHandler
  - handleRead()
  - handleWrite()
  - getInitOps()

- Acceptor
  - implements EchoReadWriteHandler
  - handleRead()
  - handleWrite()
  - getInitOps()

- ISocketReadWriteHandlerFactory
  - createHandler()

- EchoReadWriteHandlerFactory
  - createHandler()

V3

- See AsyncEchoServer/v3/*.java

Discussion on v3

- In our current implementation (Server.java)
  1. Create dispatcher
  2. Create server socket channel and listener
  3. Register server socket channel to dispatcher
  4. Start dispatcher thread

Can we switch 3 and 4?

Extending v3

- A production network server often closes a connection if it does not receive a complete request in TIMEOUT

  One way to implement time out is that
  - the read handler registers a timeout event with a timeout watcher thread with a call back
  - the watcher thread invokes the call back upon TIMEOUT
  - the callback closes the connection

  Any problem?

Extending Dispatcher Interface

- Interacting from another thread to the dispatcher thread can be tricky

  Typical solution: async command queue

  while (true) {
    - process async. command queue
    - ready events = select (or selectNow(), or select(int timeout)) to check for ready events from the registered interest events of SelectableChannels
    - foreach ready event
      - call handler
  }
**Question**

- How may you implement the async command queue to the selector thread?

```java
public void invokeLater(Runnable run) {
    synchronized (pendingInvocations) {
        pendingInvocations.add(run);
    }
    selector.wakeup();
}
```

see SelectorThread.java invokeLater

---

**Question**

- What if another thread wants to wait until a command is finished by the dispatcher thread?

```java
public void invokeAndWait(final Runnable task)
    throws InterruptedException {
    if (Thread.currentThread() == selectorThread) {
        // We are in the selector's thread. No need to schedule
        task.run();
    } else {
        // Used to deliver the notification that the task is executed
        final Object latch = new Object();
        synchronized (latch) {
            // Uses the invokeLater method with a newly created task
            this.invokeLater(new Runnable() {
                public void run() {
                    task.run();
                    // Notifies
                    synchronized(latch) {
                        latch.notify();
                    }
                }
            });
            // Wait for the task to complete.
            latch.wait();
        }
    }
    // Ok, we are done, the task was executed. Proceed.
}
```

---

**Extending v3**

- In addition to management threads, a system may still need multiple threads for performance (why?)
  - FSM code can never block, but page faults, file io, garbage collection may still force blocking
  - CPU may become the bottleneck and there maybe multiple cores supporting multiple threads (typically 2 n threads)

---

**Summary: Architecture**

- Architectures
  - Multi threads
  - Asynchronous
  - Hybrid

- Assigned reading: SEDA

---

**Problems of Event-Driven Server**

- Obscure control flow for programmers and tools
- Difficult to engineer, modularize, and tune
- Difficult for performance/failure isolation between FSMs
Another view

- Events obscure control flow
  - For programmers and tools

- Threads
  - thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
  } 

- Events
  - AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
  } 
  - RequestHandler(struct session *s) {
    ...; CacheHandler.enqueue(s);
  } 
  - CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else  ResponseHandler.enqueue(s);
  } 
  - ExitHandler(struct session *s) {
    ...;  unpin(&s);  free_session(s);  } 

State Management

- Events require manual state management
  - Hard to know when to free
    - Use GC or risk bugs

- Beyond Class: Design Patterns
  - We have seen Java as an example
  - C++ and C# can be quite similar. For C++ and general design patterns:

Summary: The High-Performance Network Servers Journey

- Avoid blocking (so that we can reach bottleneck throughput)
  - Introduce threads
    - Limit unlimited thread overhead
      - Thread pool, async io
  - Coordinating data access
    - Synchronization (lock, synchronized)
  - Coordinating behavior: avoid busy-wait
    - Wait/notify; FSM
  - Extensibility/robustness
    - Language support/design for interfaces

Some Questions

- When is CPU the bottleneck for scalability?
  - So that we need to add helpers

- How do we know that we are reaching the limit of scalability of a single machine?

- These questions drive network server architecture design

Operational Analysis

- Relationships that do not require any assumptions about the distribution of service times or inter-arrival times.
  - Identified originally by Buzen (1976) and later extended by Denning and Buzen (1978).
  - We touch only some techniques/results
    - In particular, bottleneck analysis
  - More details see linked reading
Under the Hood (An example FSM)

Operational Analysis: Resource Demand of a Request

Operational Quantities

Utilization Law

Deriving Relationship Between R, U, and S for one Device

Forced Flow Law
Bottleneck Device

Utilization $U_i = X_i S_i$

$= V_i X S_i$

$= X V_i S_i$

Define $D_i = V_i S_i$ as the total demand of a request on device $i$.

The device with the highest $D_i$ has the highest utilization, and thus is called the bottleneck.

Bottleneck vs System Throughput

Utilization $U_i = X V_i S_i \leq 1$

$\Rightarrow X \leq \frac{1}{V_{sys}}$

Example 1

- A request may need
  - 10 ms CPU execution time
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache
- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)
- Where is the bottleneck?

Example 1 (cont.)

- CPU:
  - $D_{CPU} = 10$ ms (e.g., 100 requests/s)
- Network:
  - $D_{Net} = 1$ Mbytes / 100 Mbps = 80 ms (e.g., 12.5 requests/s)
- Disk I/O:
  - $D_{disk} = 0.5 * 1$ ms * 1M/8K * 62.5 ms (e.g., 16 requests/s)

Example 2

- A request may need
  - 150 ms CPU execution time (e.g., dynamic content)
  - 1 Mbytes network bw
  - 1 Mbytes file access where
    - 50% hit in memory cache
- Suppose network bw is 100 Mbps, disk I/O rate is 1 ms per 8 Kbytes (assuming the program reads 8 KB each time)
- Bottleneck: CPU -> use multiple threads to use more CPUs, if available, to avoid CPU as bottleneck

Server Selection

- Why is the problem difficult?
- What are potential problems of just sending each new client to the lightest load server?