Effective Software

Lecture 7: Non-blocking I/O, C10K, efficient networking

David Šišlák
david.sislak@fel.cvut.cz
## Network Communication – OSI Model

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Services/Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 – Physical</strong></td>
<td>Transmits raw bit stream over physical medium.</td>
<td>Cabling/Network Interface: Binary transmission, bit rates, voltage levels, Hubs</td>
</tr>
<tr>
<td><strong>2 – Data Link</strong></td>
<td>Defines format of data on the network.</td>
<td>Switching: Switches, Bridges, Frames, PPP/SLIP, Ethernet</td>
</tr>
<tr>
<td><strong>3 – Network</strong></td>
<td>Determines how data gets from one host to another.</td>
<td>Routing: IP, IPX, ICMP, ARP, PING, Traceroute</td>
</tr>
<tr>
<td><strong>4 – Transport</strong></td>
<td>Ensures integrity of data transmission.</td>
<td>End-to-End Transport Services: TCP, UDP, SPX, AppleTalk</td>
</tr>
<tr>
<td><strong>5 – Session</strong></td>
<td>Manages connections between local and remote application.</td>
<td>Application Session Management: SQL, RPC, NFS</td>
</tr>
<tr>
<td><strong>6 – Presentation</strong></td>
<td>Formats data to be &quot;presented&quot; between application-layer entities.</td>
<td>Syntax/Semantics Layer: ASCII, JPEG, MPEG, GIF, MIDI</td>
</tr>
<tr>
<td><strong>7 – Application</strong></td>
<td>Interface to end user. Interaction directly with software application.</td>
<td>Software App Layer: FTP, HTTP, WWW, SMTP, TELNET, DNS, TFTP, NFS</td>
</tr>
</tbody>
</table>
Network Communication – Introduction

- **ports** – 16-bit number
- **IPv4** – 32-bit address
- **IPv6** – 128-bit address
  - 48-bit or more routing prefix, 16-bit or less subnet id, 64-bit interface
    - http://[1fff:0:a88:85a3::ac1f]:8080/index.html

- TCP/UDP connection identification – **quad** – src IP, src port, dst IP, dst port
C10k Problem

» handling a large number of clients (10 000s) at the same time (late 90s)
  • concurrent connections at one server requiring efficient scheduling
  • not related to requests per second
» sometime known as C1M or C10M problem (nowadays)
» approach
  • don’t use threading servers like Apache
    – each connection handled by own thread/process (pooled but limited)
    – connection operations usually use blocking operations
    – thread scheduling doesn’t scale (+cost for thread context switching)
    – thread scheduling used as packet scheduling
  • use event-driven I/O servers like Nginx
    – do packet scheduling yourself – single/multi-threaded event loop
    – using non-blocking (asynchronous) operations with event interceptors
    – multi-core scalability with controlled number of worker threads
    – reuse thread-based data structures, avoid locks (atomics, non-blocking)
Threads

» processes vs. threads
  • both support concurrent execution
  • one process has one or multiple threads
  • threads share the same address space (data and code)
  • context switching between threads is usually less expensive
  • thread inter-communication is relatively efficient using shared memory

» JVM
  • a thread executes sequence of code with own stack with frames
    t.getStackTrace()
  • own local variables
  • own method parameters

» thread creation by
  • subclass of java.lang.Thread
  • implementation of java.langRunnable
» concept of **thread pooling**
» suitable for execution of large number of asynchronous tasks
  • e.g. processing of requests in server
» reduce overhead with Thread creation for each task, context switching
» interface - `java.util.concurrent.ExecutorService`
  • `shutdown()`, `shutdownNow()`, `awaitTermination`
  • `execute(Runnable r)`
  • `Future<?> submit(Runnable r)`, `Future<T> submit(Callable<T> c)`
» `java.util.concurrent.Future<T>`
  • `boolean cancel(boolean mayInterruptIfRunning)`
  • `isCancelled()`, `isDone()`
  • `V get()`, `V get(long timeout, TimeUnit unit)`
» `java.util.concurrent.Executors` (optionally with `ThreadFactory`)
  • `newSingleThreadExecutor()`
  • `newFixedThreadPool(nThreads)`
  • `newCachedThreadPool()` – default 60 seconds keep-alive
Non-Blocking I/O Approach

» polling
  • looping to regularly check status (readiness for I/O)
  • wastes CPU cycles

» signals
  • OS generated signals on I/O readiness
  • might leave other processes inconsistent

» callbacks
  • pointer to handler function
  • stack deepening issue (callback issuing I/O)

» interrupts
  • hardware interrupts in kernel mode

» event-based
  • select
  • poll
  • epoll
Event-Based I/O - select

» select
  • defined in POSIX (Portable Operating System Interface)
  • originally used for blocking I/O
  • passed lists of descriptors cannot be reused in subsequent calls as they are modified by the system call
  • not scalable – limited descriptors + iterate over to find the event

```c
int select(int nfds, fd_set *restrict readfds, fd_set *restrict writelfds, fd_set *restrict errorfds, struct timeval *restrict timeout);

void FD_CLR(fd, fd_set *fdset);

void FD_COPY(fd_set *fdset_orig, fd_set *fdset_copy);

int FD_ISSET(fd, fd_set *fdset);

void FD_SET(fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```
Event-Based I/O - poll

» poll
  • polled descriptors not limited
  • descriptors can be reused
  • better but you still need iterate over descriptors to find events

```c
int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

```c
struct pollfd {
    int    fd;       /* file descriptor */
    short  events;  /* events to look for */
    short  revents; /* events returned */
};
```
Event-Based I/O - epoll

» epoll
  • Linux only (e.g. Windows has IOCP – IO Completion Ports)
  • scalable
  • monitored events can be modified while polling (via syscall)
  • returns triggered events directly

» API
  • epoll_create & epoll_create1 – initialize epoll instance (kernel structure)
  • epoll_ctl – add/modify/remove descriptors to epoll instance
  • epoll_wait – wait for events up to timeout

» modes
  • level triggered – wait always returns if event is available
  • event triggered (EPOLLET) – readiness returned upon incoming event only
    (you have to process all pending events before next wait !)

» events
  • EPOLLIN, EPOLLOUT, EPOLLPRI
  • EPOLLRDHUP, EPOLLHUP
  • EPOLLERR
Epoll Usage

epoll structure:

```c
typedef union epoll_data
{
    void   *ptr;
    int     fd;
    __uint32_t  u32;
    __uint64_t  u64;
} epoll_data_t;

struct epoll_event
{
    __uint32_t  events;  /* Epoll events */
    epoll_data_t data;  /* User data variable */
};
```

initialization:

```c
int epfd = epoll_create1(0);
...
struct epoll_event ev;
int client_sock;
...
ev.events = EPOLLIN | EPOLLPRI | EPOLLERR | EPOLLHUP;
ev.data.fd = client_sock;
int res = epoll_ctl(epfd, EPOLL_CTL_ADD, client_sock, &ev);
```
while (1) {
    // wait for something to do...
    int nfds = epoll_wait(epfd, events,
                           MAX_EPOLL_EVENTS_PER_RUN,
                           EPOLL_RUN_TIMEOUT);

    if (nfds < 0) die("Error in epoll_wait!");

    // for each ready socket
    for(int i = 0; i < nfds; i++) {
        int fd = events[i].data.fd;
        handle_io_on_socket(fd);
    }
}
JAVA Networking – TCP Client

» **Socket**
  • client end-point of network TCP/IP connection
  • is bound to particular destination IP and port
  • each TCP/IP connection is uniquely identified by its two end-points
  • provides input/output streams

```java
try {
    Socket echoSocket = new Socket(host: "localhost", port: 7);
    PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), autoFlush: true);
    BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

    String userInput;

    while ((userInput = stdin.readLine()) != null) {
        out.println(userInput);
        System.out.println("echo: " + in.readLine());
    }
}
```
JAVA Networking – TCP Server

» ServerSocket
  • special socket representing listening TCP/IP end-point
  • within constructor you specify the port, and optionally IP where it has to be bound
  • wait for establishing connection using method Socket accept()
threading server example – each handler runs in own thread with blocking I/O

```java
ExecutorService clientRunner = Executors.newCachedThreadPool();
try {
   ServerSocket serverSocket = new ServerSocket( port: 7)
} {
   while (true) {
      final Socket s = serverSocket.accept();
      clientRunner.execute() -> {
         try {
            BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
            PrintWriter out = new PrintWriter(s.getOutputStream(), autoFlush: true)
            String line;
            while (s.isConnected()) {
               if ((line = in.readLine()) != null) {
                  out.println(line);
               }
            }
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
} catch (Exception e) {
   e.printStackTrace();
} finally {
   clientRunner.shutdownNow();
}
```
» **DatagramPacket**
  - independent, self-contained message sent over the network
  - like packet
    - InetAddress address, int port – destination
    - byte data[], int length, int offset
    - SocketAddress sa – sender

» **DatagramSocket**
  - sending or receiving point for a packet delivery service
  - can be bound to any available port (using default constructor)
  - connect(InetAddress, int) – can sent or receive packets only specified host, if not set in DatagramPacket automatically fill
  - send(DatagramPacket p), receive(DatagramPacket p) – blocking IO

» **MulticastSocket**
  - additional capabilities for joining/leaving multicast groups, loopback
  - multicast IP (IGMP – Internet Group Management Protocol)
    - 224.0.0.0 – 239.255.255.255
» **scalable I/O** – asynchronous I/O requests and polling

» high-speed **block-oriented** binary and character I/O working – including mapping files to the memory, using channels and selectors

» Channel is like a block device working with Buffers
java.nio.Buffer

- **linear, finite sequence of elements** of a specific primitive type
  - ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer, MappedByteBuffer {FileChannel.map(...)}
- not thread safe, **multi mode** for the same buffer (both read & write)
- key properties – $0 \leqslant \text{mark} \leqslant \text{position} \leqslant \text{limit} \leqslant \text{capacity}$
  - capacity – numbers of elements, never changing!
  - limit – index of the first element that should not be read or written
  - position – index of the next element to be read or written
  - mark – index to which its position is set after reset()
- clear() – position=0, limit=capacity => ready for channel read (put)
- flip() – limit=position, position=0 => ready for channel write (get)
- rewind() – limit unchanged, position=0 => ready for re-reading
- mark() – mark = position
- reset() – position=mark
» write mode – channel.read(buf); buf.put(...);
» read mode – channel.write(buf); ... buf.get();
» java.nio.Buffer
  • isReadOnly() – can be read-only
  • hasArray() – is backed by an accessible array (array())
  • equals(), compareTo() – compare remainder sequence

• can be allocated to native memory (see next slide)

• typical usage

1. Write data into the Buffer
2. Call buffer.flip()
3. Read data out of the Buffer
4. Call buffer.clear() or buffer.compact()

Note: compact() – bytes between position and limit are copied to the beginning of the buffer and prepare for writing again
JVM – Memory Layout – Native Memory

Stack

Thread
- Program Counter
  - Stack
  - Native Stack

Non Heap
- Other native memory
- Metaspace
  - Code Cache
  - Method Area

Heap
- Young Generation
- Old / Tenured Generation
  - Eden Space
  - Survivor Spaces
  - Minor Garbage Collection
  - Major Garbage Collection

JNI
native
NIO
buffers

native memory
ByteBuffer.allocateDirect(...)  
stored out of JAVA heap in native memory  
allow native code and Java code to share data without copying  
  • useful for file and socket  
    – the same memory is passed to kernel during calls  
multiple buffers can share native memory  
  • slice()/duplicate() – independent position, limit, mark, shared content  
  • asReadOnlyBuffer() – read only view of shared content  
tuning/tracking  
  • -XX:MaxDirectMemorySize=N (default unlimited)  
  • -XX:NativeMemoryTracking=off|summary|detail  
  • -XX:+PrintNMTStatistics

Note: usage of heap buffers implies content copy out/in Java heap space due to possible relocations by GC
» **one thread** works with **multiple channels at the same time**
  
  • **epoll-based** if OS support epoll

» **Channel** – cover UDP+TCP network IO, file IO
  
  • FileChannel – from Input/OutputStream or RandomAccessFile
  • DatagramChannel
  • MulticastChannel (since 1.7)
  • SocketChannel
  • ServerSocketChannel
» Channel
  • read/write at the same time (streams are only one-way)
  • always read/write from/to a buffer

» FileChannel
  • only blocking
  • support – direct buffers, mapped files, locking
  • bulk transfers between channels
    - no copy at all, direct transfer e.g. to socket
    - transferFrom(sourceChannel, int pos, int count)
    - transferTo(int pos, int count, dstChannel)
» **SocketChannel** – client end-point of TCP/IP
  • can be configured as non-blocking before connecting
  • `SocketChannel socket.getChannel();`
  • `SocketChannel SchSocketChannel.open();`
  • `sch.connect(...)`

  • `write(...) and read(...)` may return without having written/read anything for non-blocking channel

» **ServerSocketChannel** – server end-point of TCP/IP
  • can be configured as non-blocking
  • can be created directly using `open()` or from `ServerSocket`
  • `accept()` – returns `SocketChannel` in the same mode
» **Selector**
  
  • Selector Selector.open();
  • only channels in **non-blocking** mode can be registered
    
    channel.configureBlocking(false);
    
    SelectionKey channel.register(selector, SelectionKey.OP_READ);
  
  • FileChannel doesn’t support non-blocking mode

» **SelectionKey** – events you can listen for (can be combined together)
  
  • OP_CONNECT
  • OP_ACCEPT
  • OP_READ
  • OP_WRITE

» events are filled by channel which is ready with operation
SelectionKey – returned from register method

- interest set – your configured ops
- ready set – which ops are ready, sk.isReadable(), sk.isWritable(), ...
- channel
- selector
- optional attached object – sk.attach(Object obj);
  Object sk.attachment()
  SelectionKey channel.register(selector, ops, attachmentObj);
Selector with registered one or more channels

- `int select()` – blocks until at least one channel is ready
- `int select(long timeout) – with timeout milliseconds`
- `int selectNow()` – doesn’t block at all, returns immediately

```
return the number of channels which are ready from the last call
Set<SelectionKey> selector.selectedKeys();
```

- `wakeup()` – different thread can “wake up” thread blocked in `select()
- `close()` – invalidates selector, channels are not closed
JAVA – NIO Server – Using Multiple Reactors

Client

NIOReactors

NIOAcceptor Handler

Client

Client

NIOClientHandlers

receive

parse

send

Queued parse tasks

ThreadPool

Worker thread

Worker thread
public class NIOServer {
    final static int MSG_SIZE = 1_000_000;
    private final static NIOResultor[] reactors;
    static ExecutorService workers = Executors.newWorkStealingPool();
    static BufferPool bufferPool = new BufferPool();

    static {
        reactors = new NIOResultor[4];
        try {
            for (int i=0; i<reactors.length; i++) {
                reactors[i] = new NIOResultor();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void startNIOServer(int port) throws IOException {
        reactors[0].register(NIOAcceptorHandler.getNIOAcceptorHandler(reactors, port));
    }

    static class BufferPool {...}
}
```java
import java.nio.channels.Selector;
import java.util.concurrent.ConcurrentLinkedQueue;

public class NIOReactor implements Runnable {
    private final Selector s = Selector.open();
    private final ConcurrentLinkedQueue<Runnable> toRegister = new ConcurrentLinkedQueue<>();

    NIOReactor() throws IOException {
        Thread t = new Thread( Runnable::run );
        t.setDaemon( true );
        t.start();
    }

    void register(Runnable target) {
        toRegister.add( target );
        s.wakeup();
    }  

    @Override
    public void run() {
        try {
            while ( true ) {
                try {
                    s.select();
                    for (SelectionKey key : s.selectedKeys()) {
                        if (key.attachment() != null) ((Runnable) key.attachment()).run();
                    }
                    s.selectedKeys().clear();
                    NIOHandler t;
                    while ((t = toRegister.poll()) != null) {
                        t.setSelectionKey(s, t.getInitialSelectableOps(), t);
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
abstract class NIOHandler implements Runnable {
    private final SelectableChannel selectableChannel;
    private final int initialSelectableOps;
    private SelectionKey selectionKey = null;

    NIOHandler(SelectableChannel selectableChannel, int initialSelectableOps) {
        this.selectableChannel = selectableChannel;
        this.initialSelectableOps = initialSelectableOps;
    }

    SelectableChannel getSelectableChannel() {
        return selectableChannel;
    }

    int getInitialSelectableOps() {
        return initialSelectableOps;
    }

    SelectionKey getSelectionKey() {
        return selectionKey;
    }

    void setSelectionKey(SelectionKey selectionKey) {
        this.selectionKey = selectionKey;
    }
}
class NIOAcceptorHandler extends NIOHandler {
    private final NIOReactor[] reactors;
    private final ServerSocketChannel ssch;
    private int roundRobin = 0;

    static NIOAcceptorHandler getNIOAcceptorHandler(NIOReactor[] reactors, int port) throws IOException {
        ServerSocketChannel ssch = ServerSocketChannel.open();
        ssch.socket().bind(new InetSocketAddress(port));
        ssch.configureBlocking(true: false);
        return new NIOAcceptorHandler(reactors, ssch, SelectionKey.OP_ACCEPT);
    }

    private NIOAcceptorHandler(NIOReactor[] reactors, ServerSocketChannel ssch, int selectableOps) {
        super(ssch, selectableOps);
        this.reactors = reactors;
        this.ssch = ssch;
    }

    @Override
    public void run() {
        try {
            SocketChannel sch = ssch.accept();
            if (sch != null) {
                reactors[roundRobin].register(new NIOClientHandler(sch));
                roundRobin = (roundRobin+1)%reactors.length;
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```java
class NIOClientHandler extends NIOHandler {
    private final SocketChannel socketChannel;
    private ByteBuffer readBuf;
    private ByteBuffer writeBuf = null;

    NIOClientHandler(SocketChannel socketChannel) throws IOException {
        super(socketChannel, SelectionKey.OP_READ);
        this.socketChannel = socketChannel;
        readBuf = NIOServer.bufferPool.getBuffer();
        socketChannel.configureBlocking(block: false);
    }

    @Override
    public void run() {
        try {
            if (getSelectionKey().isReadable()) read();
            else if (getSelectionKey().isWritable()) write(setWriteInterest: false);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    private void read() throws IOException {...}

    private void process() {...}

    private void write(boolean setWriteInterest) throws IOException {...}
}
```
```java
private void read() throws IOException {
    if (socketChannel.read(readBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (readBuf.remaining() == 0) {
        getSelectionKey().interestOps(0);
        getSelectionKey().selector().wakeup();
        NIOServer.workers.execute(this::process);
    }
}

private void process() {
    try {
        readBuf.flip();
        writeBuf = NIOServer.bufferPool.getBuffer();

        // DO processing and prepare data in writeBuf
        writeBuf.put(readBuf);
        writeBuf.flip();

        NIOServer.bufferPool.releaseBuffer(readBuf);
        readBuf = null;
        write( setWriteInterest: true );
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```
private void write(boolean setWriteInterest) throws IOException {
    if (socketChannel.write(writeBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (writeBuf.remaining() > 0) {
        if (setWriteInterest) {
            getSelectionKey().interestOps(SelectionKey.OP_WRITE);
        }
    } else {
        readBuf = writeBuf;
        readBuf.clear();
        writeBuf = null;
        getSelectionKey().interestOps(SelectionKey.OP_READ);
    }
    if (setWriteInterest) {
        getSelectionKey().selector().wakeup();
    }
}
private void write(boolean setWriteInterest) throws IOException {
    if (socketChannel.write(writeBuf) == -1) {
        getSelectionKey().cancel();
        socketChannel.close();
    } else if (writeBuf.remaining() > 0) {
        if (setWriteInterest) {
            getSelectionKey().interestOps(SelectionKey.OP_WRITE);
        }
    } else {
        readBuf = writeBuf;
        readBuf.clear();
        writeBuf = null;
        getSelectionKey().interestOps(SelectionKey.OP_READ);
    }
    if (setWriteInterest) {
        getSelectionKey().selector().wakeup();
    }
}