# Návrh a vývoj obchodovacích systémů

29. dubna 2015


# Agenda

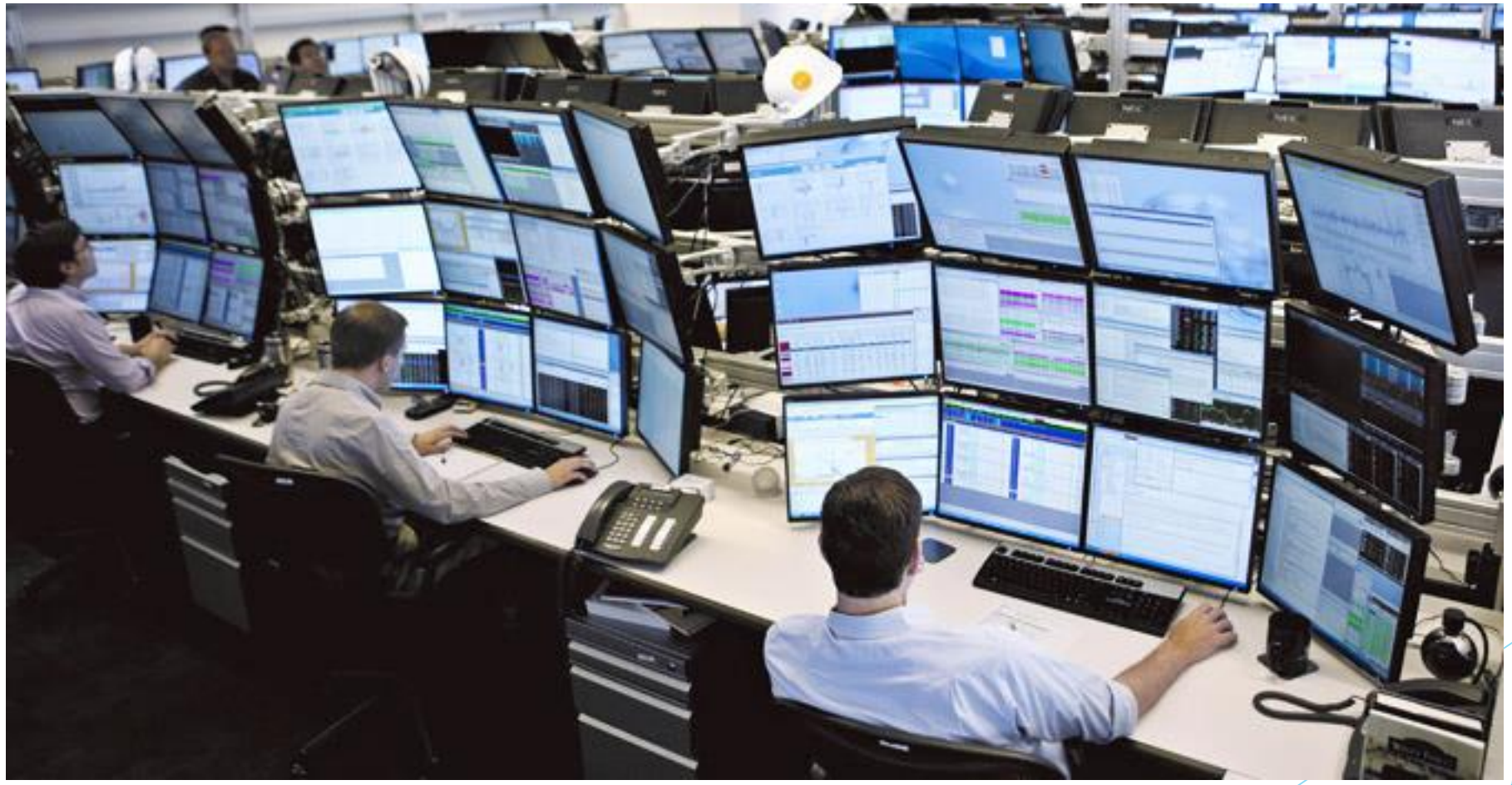
- Základní principy obchodování na burzách cenných papírů
- Využití znalostí z MFF
- Zásady vývoje real-time obchodovacích systémů
- Demo platformy DET Hub
- Prostor pro vaše otázky

# Market Depth

| Vol | Bid | Offer | Vol |
|---|---|---|---|
| 256 | 10.5 | 10.8 | 5 |
| 15 | 10.2 | 10.9 | 154 |
| 1000 | 9.9 | 10.10 | 12 |
| | | 10.12 | 20 |
| | | 10.15 | 900 |

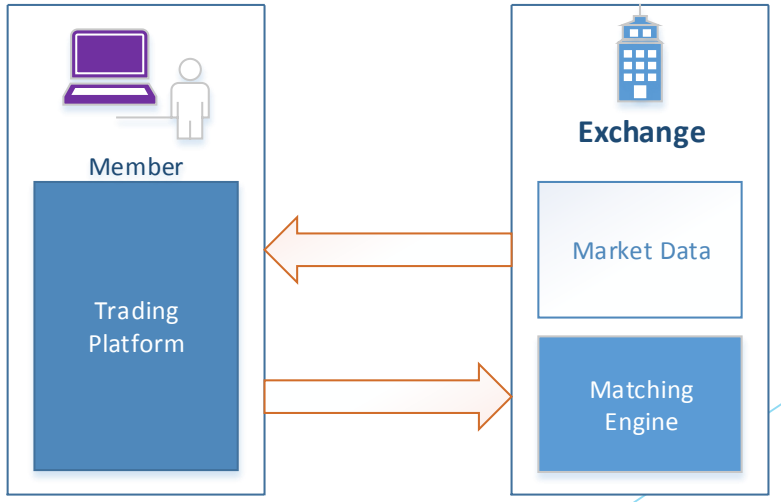
# Simple Trading Screen

# Trading Floor



# Napojení na burzu

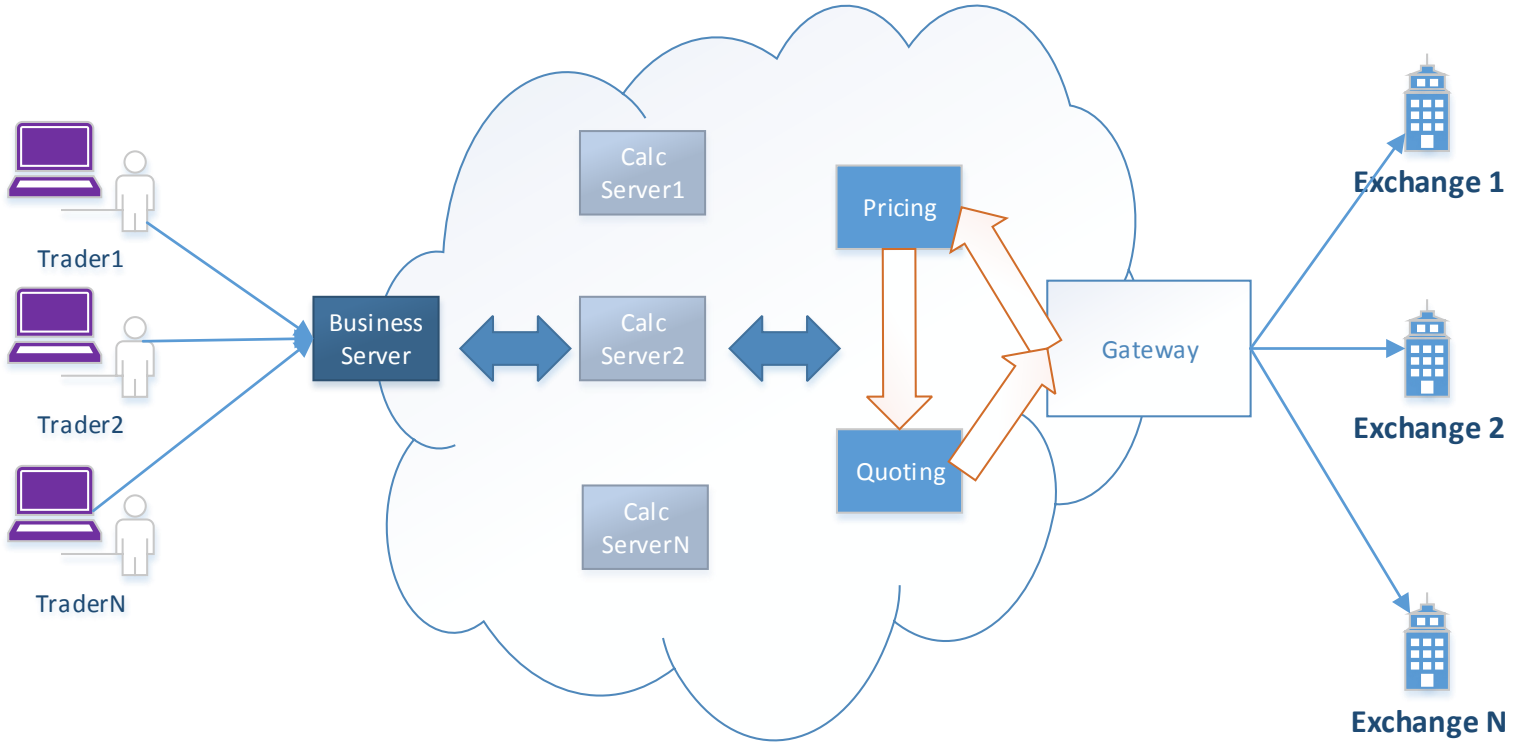
- Každá burza nabízí rozhraní pro
  - Příjem orderů a kotací
  - Market data
- Protokoly
  - Komunikace založená na výměně zpráv
  - Binární
    - ETS, OUCH
  - Textové
    - FIX
  - XML
    - Arena



# Vlastnosti obchodovacích systémů

- Low-latency: Důraz kladen na rychlost odezvy systémů (mikrosekundy)
- Low-jitter: stabilita zpracování událostí
- High-throughput: Zpracování tisíců událostí za vteřinu
- Robustnost
  - High Availability
  - Fast Recovery

# Příklad


Trader1
Trader2
TraderN
Business Server
Calc Server1
Calc Server2
Calc ServerN
Pricing
Quoting
Gateway
Exchange 1
Exchange 2
Exchange N

# Užitečné kurzy na MFF

- Teoretické přednášky
  - Algoritmy
  - Datové struktury
  - Složitost
- Praktické přešdnášky
  - Operační systémy I, II
  - Middleware
  - Programování v UNIXu

# Zásady vývoje real-time obchodovacích systémů

- Multi-threading
  - Asynchronní programování
  - Actor model
- Paměť
  - Garbage collection
  - Cache

# Asynchronní vs synchronní přístup

## Synchronní volání

```
public void sync() {
    List<Item> items = load();
    for (Item item : items) {
        process(item);
    }
}
```

## Asynchronní volání

```
public void async() {
    loadAsync(callback);
}

private final Callback callback =
        new Callback() {
    @Override
    public void apply(List<Item> items) {
        for (Item item : items) {
            process(item);
        }
    }
};
```

# Zámky

## 2 vlákna (zámek)

```
synchronized (sync) {
    if (start == 0) {
        start = System.currentTimeMillis();
    }
}

while (true) {
    int index;
    synchronized (sync) {
        if ((index = counter++) >= COUNT) break;
    }
    double squareRoot = Math.sqrt(index);
    synchronized (sync) {
        roots[index] = squareRoot;
    }
}

synchronized (sync) {
    if (time == 0) {
        time = System.currentTimeMillis() - start;
    }
}
```

## 1 vlákno

```
long start = System.currentTimeMillis();
for (int i = 0; i < COUNT; ++i) {
    roots[i] = Math.sqrt(i);
}
time = System.currentTimeMillis() - start;
```

# Zámky II

```
public class State {
    private int traded;

    public int getTraded() { return traded; }
    public void increaseTraded(int value) { traded += value; }
}

for (int i = 0; i < 100_000_000; ++i) {
    synchronized (state) {
        state.increaseTraded(1);
    }
}

for (int i = 0; i < 100_000_000; ++i) {
    synchronized (state) {
        if (state.getTraded() > 100_000_000) {
            System.out.println("Limit exceeded");
        }
    }
}
```

# Mutable vs Immutable objects

## Mutable

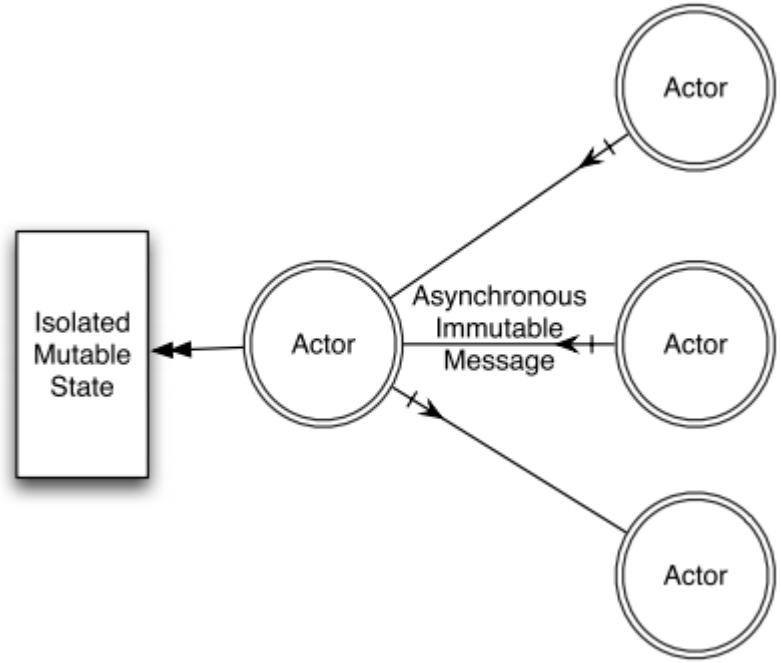
```
public class State {
    private int traded;

    public int getTraded() {
        return traded;
    }

    public void increaseTraded(int value) {
        traded += value;
    }
}
```

## Immutable

```
public class ImmutableState {
    private final int traded;

    public ImmutableState(int traded) {
        this.traded = traded;
    }

    public ImmutableState increaseTraded(int value) {
        return new ImmutableState(traded + value);
    }

    public int getTraded() {
        return traded;
    }
}
```

# Actor Model

- The **actor model** in computer science is a mathematical **model** of concurrent computation that treats "**actors**" as the universal primitives of concurrent computation: in response to a message that it receives, an **actor** can make local decisions, create more **actors**, send more messages, and determine how to respond ...

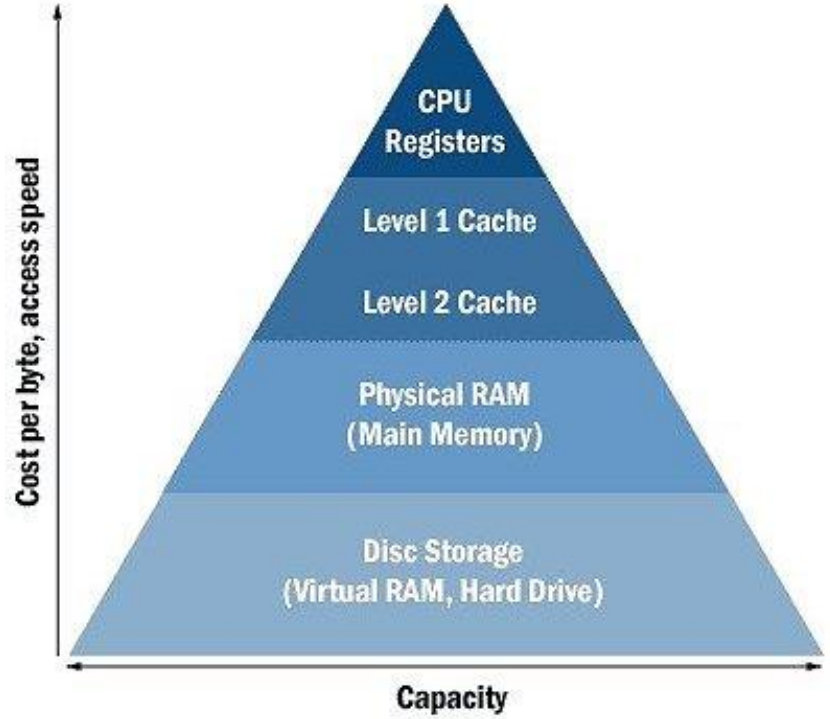


# Paměť

- Rychlost paměti je řádově nižší než rychlost CPU
  - Optimalzace datových struktur = zvýšení výkonu
- Vyšší programovací jazyky ulehčují práci s pamětí
  - Garbage Collection

# Memory Cache

- Přístup do hlavní paměti je řádově pomalejší než do registrů a L1, L2, L3 cache
- CPU nahrává data z hlavní paměti do cache v tzv. cache lines (typicky 64 bytů)



# Memory cache příklad

- Přidej 10 milionů integerů do listu
- 10 milion krát odeber první prvek a dej ho nakonec

```
private static void listTest() {
    List<Integer> list = new ListImpl<>();
    long start = System.currentTimeMillis();
    for (int i = 0; i < 10_000_000; ++i) {
        list.add(i);
    }
    System.out.println(System.currentTimeMillis() - start);
    for (int i = 0; i < 10_000_000; ++i) {
        int head = list.remove();
        list.add(head);
    }
    System.out.println(System.currentTimeMillis() - start);
}
```

# LinkedList vs RingBuffer

## LinkedList

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode =
        new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

## RingBuffer

```
public class RingBuffer {
    private final int[] array;

    private int length;
    private int start;

    public RingBuffer(int length) {
        array = new int[length];
    }

    public int remove() {
        int res = array[start];
        start = (start + 1) % array.length;
        --length;
        return res;
    }

    public void add(int i) {
        array[length++] = i;
    }
}
```

# Flyweight Pattern

- Optimalizace cache miss problému

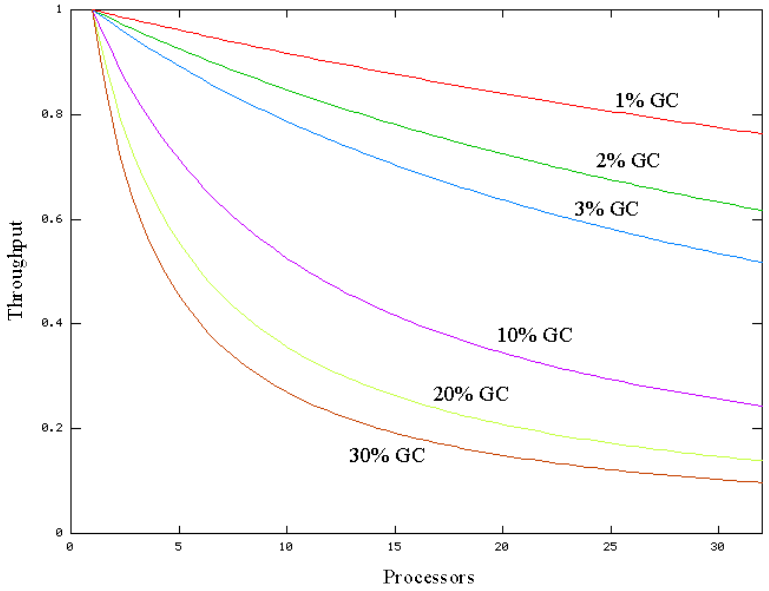
```
private static class Trade {

    private static final ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 1024 * 1024);
    private static final Trade flyweight = new Trade();

    private static int offset = 0;
    private static final int priceOffset = offset += 0;
    private static final int quantityOffset = offset += 8;
    private static final int sideOffset = offset += 8;
    private static final int objectSize = offset += 2;
    private int objectOffset;

    public static int getObjectSize() { return objectSize; }

    void setObjectOffset(final int objectOffset) { this.objectOffset = objectOffset; }

    public long getPrice() { return buffer.getLong(objectOffset + priceOffset); }

    public void setPrice(final long price) { buffer.putLong(objectOffset + priceOffset, price); }

    public long getQuantity() { return buffer.getLong(objectOffset + quantityOffset); }

    public void setQuantity(final long quantity) { buffer.putLong(objectOffset + quantityOffset, quantity); }

    public char getSide() { return buffer.getChar(objectOffset + sideOffset); }

    public void setSide(final char side) { buffer.putChar(objectOffset + sideOffset, side); }

    private static Trade create(final int index) {
        final int offset = (index * getObjectSize());
        flyweight.setObjectOffset(offset);
        return flyweight;
    }
}
```

# Flyweight Pattern – použití

```
public static void main(final String[] args) throws Exception {

    final long start = System.nanoTime();
    for (int i = 0; i < 1000_000; i++) {
        Trade trade = Trade.create(i);
        trade.setPrice(i);
        trade.setQuantity(i);
        trade.setSide((i & 1) == 0 ? 'B' : 'S');
    }

    long buyCost = 0;
    long sellCost = 0;
    for (int i = 0; i < 1000_000; i++) {
        final Trade trade = Trade.create(i);
        if (trade.getSide() == 'B') {
            buyCost += (trade.getPrice() * trade.getQuantity());
        } else {
            sellCost += (trade.getPrice() * trade.getQuantity());
        }
    }
    long duration = System.currentTimeMillis() - start;
    System.out.println("duration " + duration + "ms");
    System.out.println("buyCost = " + buyCost + " sellCost = " + sellCost);
}
```

# Garbage Collector

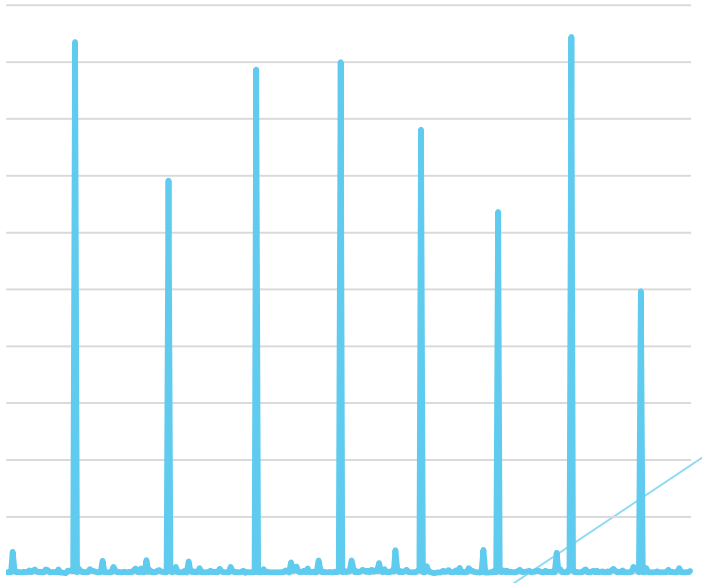
- Výkon komplexních systémů je výrazně ovlivněn délkou Garbage collection
- Ztráta výkonu 2% na 1 procesoru = ztráta výkonu 40% na 32 procesorech



# Garbage Collector II

- Standardní implementace JVM a CLR používají stop-the-world garbage collection
- Všechny thready zastaveny na potenciálně velmi dlouhý časový interval

```
for (int i = 0; i < 100_000_000; ++i) {
    long start = System.nanoTime();
    map.add(UUID.randomUUID().toString());
    int time = (int) (System.nanoTime() - start);

    if (time > max) max = time;

    if (map.size() > 100_000) map.remove(map.first());
    if (i > 0 && i % 10_000 == 0) {
        System.out.println(max);
        max = 0;
    }
}
```



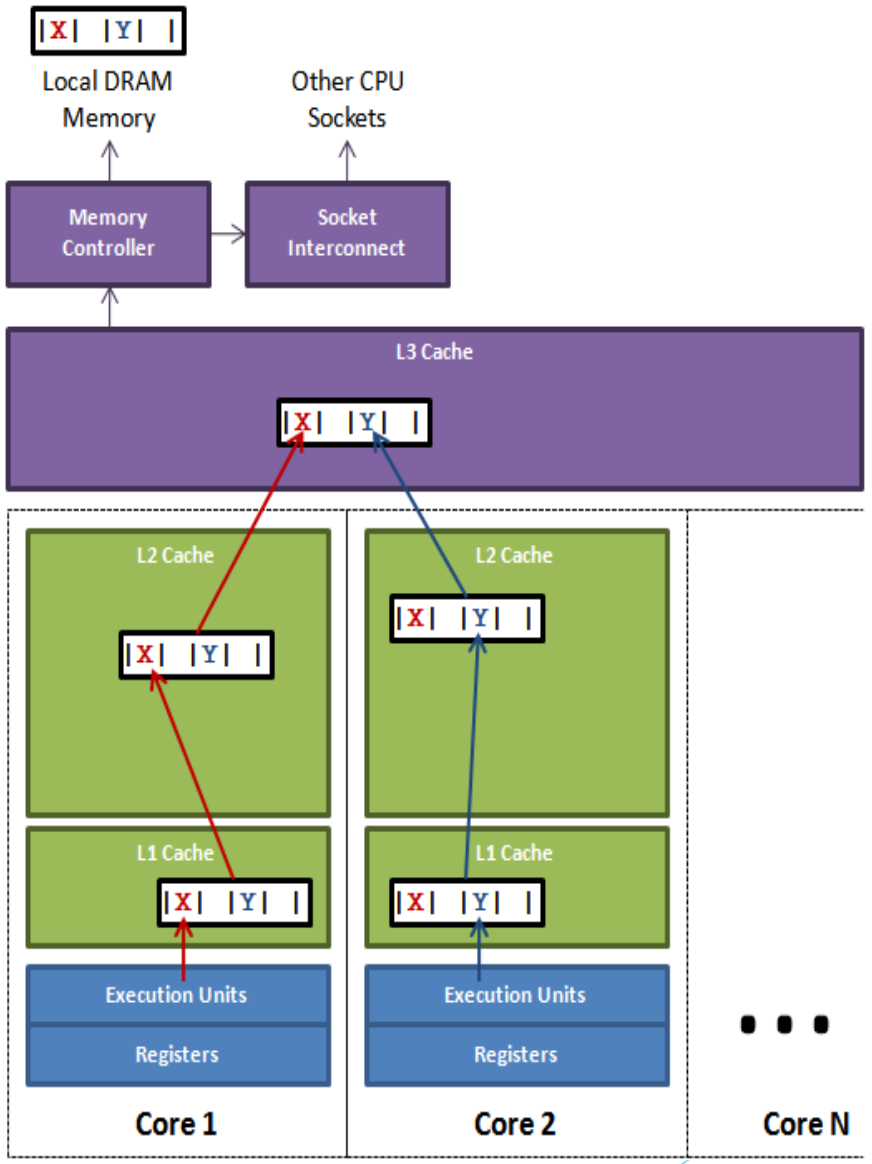
# Prevence Stop-The-World

- Použití komerční JVM, která nepodléha STW a zajišťuje continuous garbage collection – Azul Zing
- Použití programovacích technik, které zajišťují, že během vykonávání programu se nevytváří žádné nebo naprosté minimum objektů

- Object Pools
  - předvytvořená množina objektů, které se používají stále dokola
  - klient požádá pool o objekt, provede s ním operace, které potřebuje a vrátí ho zpět do poolu
- Offheap Structures
  - Alokace objektů v paměti mimo heap, která nespadá pod GC management
  - Vlastní management paměti (alokace/dealokace) jako v C
  - V JVM využití sun.misc.unsafe

# False Sharing

- Vzniká v případech kdy dvě či více vláken modifikují vzájemně nezávislé proměnné, které sdílí stejnou cache line
- Na první pohled obtížné odhalit
- Je potřeba znát Memory Layout
- Řešení:
  - padding
  - Související data v souvisejících blocích pamětí



# False Sharing příklad

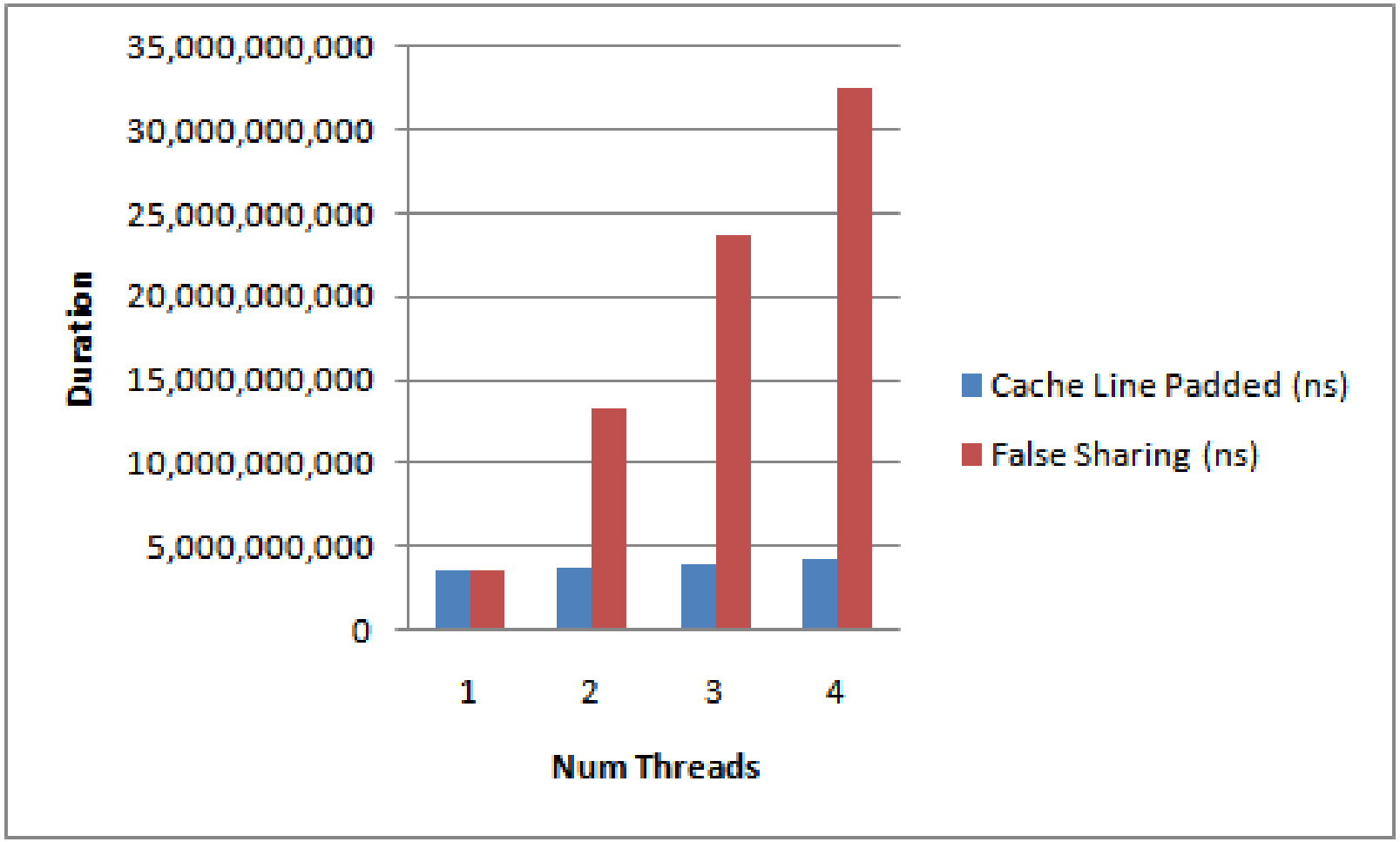
- Spusť N vláken a v každém vlákně 500 milion krát změň nezávislou proměnnou

```
public final static class VolatileLong {
    public volatile long value = 0L;
    public long p1, p2, p3, p4, p5, p6; // comment out to disable padding
}
```

```
public final static int NUM_THREADS = 4;
public final static long ITERATIONS = 500L * 1000L * 1000L;
private final int arrayIndex;
private static VolatileLong[] longs = new VolatileLong[NUM_THREADS];
static {
    for (int i = 0; i < longs.length; i++) {
        longs[i] = new VolatileLong();
    }
}

public FalseSharing(final int arrayIndex) {
    this.arrayIndex = arrayIndex;
}

public static void main(final String[] args) throws Exception {
    final long start = System.nanoTime();
    runTest();
    System.out.println("duration = " + (System.nanoTime() - start));
}
```

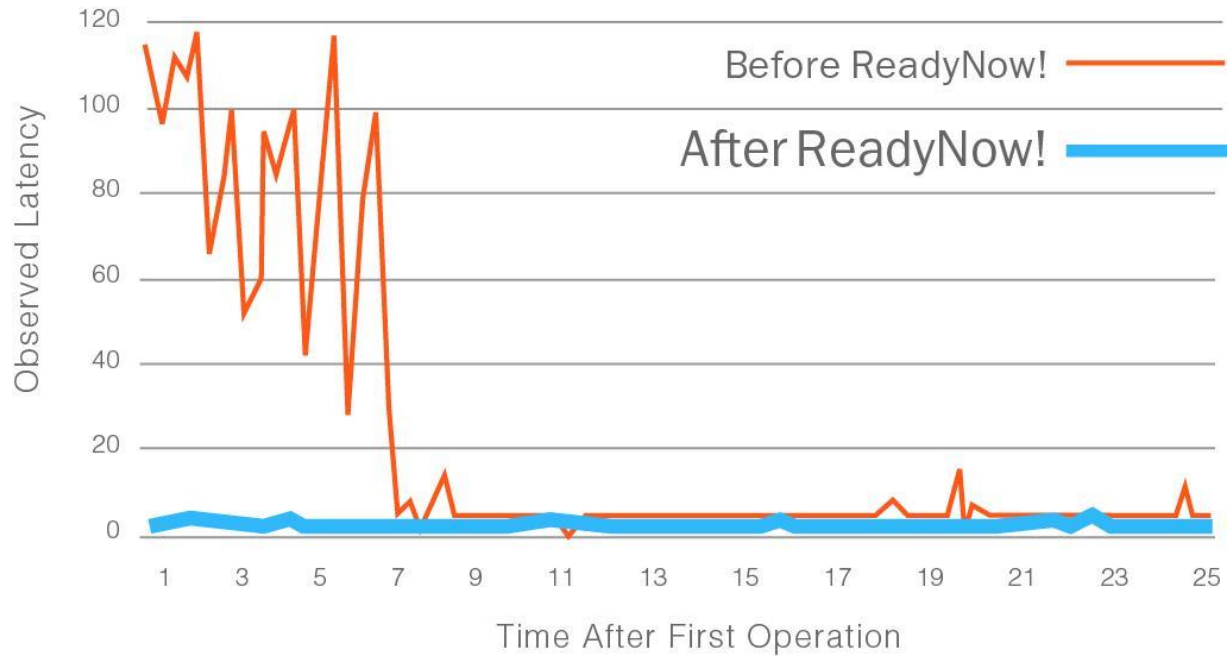
```
private static void runTest() throws InterruptedException {
    Thread[] threads = new Thread[NUM_THREADS];
    for (int i = 0; i < threads.length; i++) {
        threads[i] = new Thread(new FalseSharing(i));
    }
    for (Thread t : threads) {
        t.start();
    }
    for (Thread t : threads) {
        t.join();
    }
}

public void run() {
    long i = ITERATIONS + 1;
    while (0 != --i) {
        longs[arrayIndex].value = i;
    }
}
```

# False Sharing výsledky



# Warm-Up

- Standardní implementace JVM a CLR používají JIT a run-time optimalizace kódu
- Systémy po startu vykazují sníženou výkonost



# Výběr vhodného HW

- dostatek jader – dodržovat pravidlo “core per application thread” + jádra pro obhospodaření OS
- ideálně 2-socket řešení ... celý OS běží na jednom socketu a aplikace samotná na socketu druhém – viz OS Tunning
- CPU s co nejvyšší nominální + turboboost frekvencí
- CPU s co největší cache
- Dostatek operační paměti
- Oddělené SSD disky pro OS a aplikaci samotnou
- Použití High-performance síťových karet - např SolarFlare
  - umožňují preskočení OS Kernelu při zpracování síťového přenosu
- Zajištění, aby sběrnice, ve které je síťová karta byla na stejném NUMA node jako Socket, který bude přiřazen dané aplikaci

# OS Tunning

- Pro docílení nejvyššího a především stabilního výkonu je nezbytné se při nasazování systému zaměřit také na samotný operační systém
- Často se jedná o několikadenní proces, který je závislý na použitém HW a OS, tudíž neexistuje jednoznačný a všudefungující postup
- Izolování jader pro danou aplikaci ... taskset, isolcpu, numactl
- Zakázání IRQ přerušení na jádrech, na kterých beží aplikace ... irqbalance
- Zajištění, že všechny systemové procesy a služby běží na jádrech, na kterých neběží aplikace samotná
- Priřazení jednotlivých aplikačních vláken daným jádrum ... zachování pravidla “thread per core” – taskset nebo Java-Thread-Affinity knihovna
- Správné nastavení kernel parametrů - vm.min_free_kbytes, vm.swappiness, transparent huge pages a další
- Diagnostika jitteru: jHiccup (Azul), sysjitter (SolarFlare)

# OS Tunning příklad – CPU affinity

- Posílání multicast zpráv přes “dummy” interface – zajistí, že samotná síť nebude ovlivňovat výsledky
- Sender:

```
public static void main(final String[] args) throws Exception {
    final byte[] buffer = ("This is a string with sufficient"
            + " data to test a packet sending").getBytes("ASCII");
    // Create socket and packet objects
    final Thread t = new Thread(new MultiCastSender());
    t.start();
    while (messageCounter++ < count)
    {
        packet.setData(buffer);
        sendSocket.send(packet);
    }
    sendSocket.close();
    t.interrupt();
    t.join();
}
```

```
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        try {
            Thread.sleep(1000L);
        } catch (InterruptedException ex) {
            break;
        }
        final long newTimestamp = System.currentTimeMillis();
        final long duration = newTimestamp - lastTimestamp;
        final long newMessageCounter = messageCounter;
        final long numberOfMessages = newMessageCounter - lastMessageCounter;
        System.out.format("Sent %d messages in %dms\n",
                Long.valueOf(numberOfMessages), Long.valueOf(duration));
        lastTimestamp = newTimestamp;
        lastMessageCounter = newMessageCounter;
    }
}
```

# OS Tunning příklad – CPU affinity

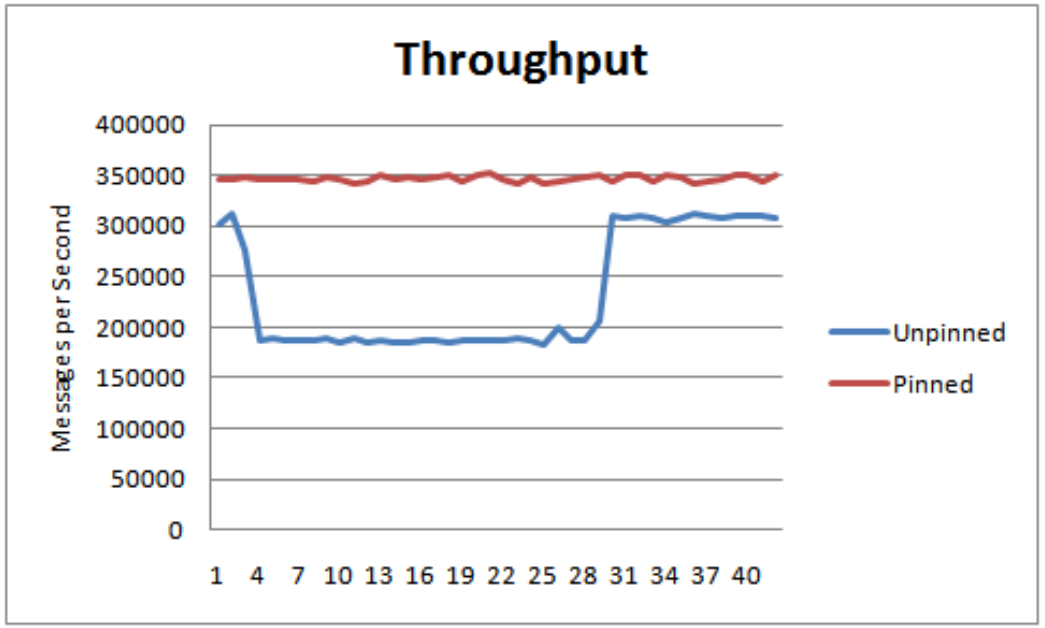
- Receiver:

```
public static void main(final String[] args) throws Exception {

    final int multiCastPort = 4447;
    final String address = args[0];
    final NetworkInterface networkInterface = NetworkInterface
            .getByName(args[1]);
    final MulticastSocket receiveSocket = new MulticastSocket(multiCastPort);
    final SocketAddress socketAddress = new InetSocketAddress(address, multiCastPort);
    receiveSocket.joinGroup(socketAddress, networkInterface);
    final byte[] buffer = new byte[BUFFER_SIZE];
    final DatagramPacket packet = new DatagramPacket(buffer, BUFFER_SIZE);
    new Thread(new MultiCastReceiver()).start();
    while (true) {
        packet.setLength(BUFFER_SIZE);
        receiveSocket.receive(packet);
        ++messageCounter;
    }
}
```

```
public void run() {
    while (true) {
        try {
            Thread.sleep(1000L);
        } catch (InterruptedException ex) {
            break;
        }
        final long newTimestamp = System.currentTimeMillis();
        final long duration = newTimestamp - lastTimestamp;
        final long newMessageCounter = messageCounter;
        final long numberOfMessages = newMessageCounter
                - lastMessageCounter;
        System.out.format("Received %d messages in %dms\n",
                Long.valueOf(numberOfMessages), Long.valueOf(duration));
        lastTimestamp = newTimestamp;
        lastMessageCounter = newMessageCounter;
    }
}
```

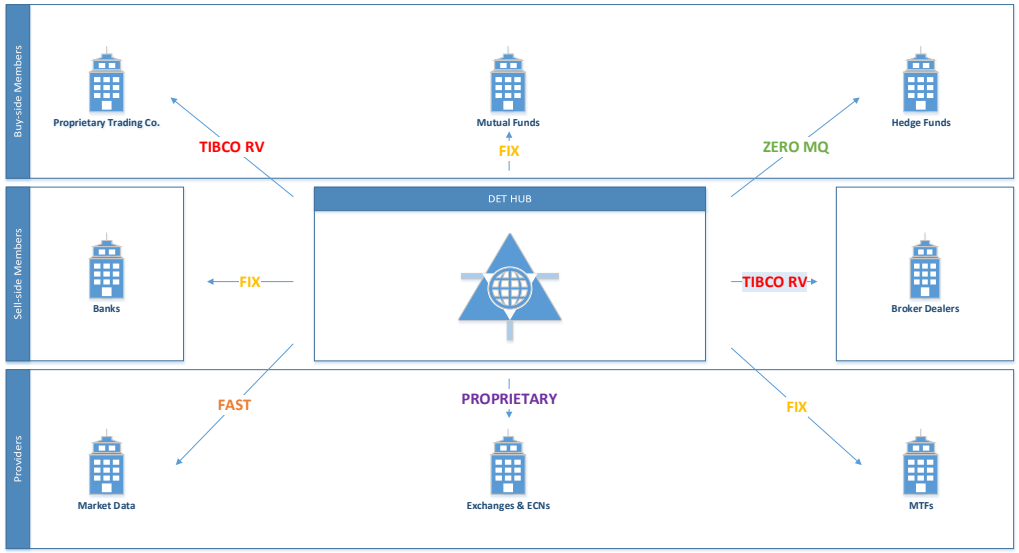
# OS Tunning příklad – výsledky

- Spuštění bez použití CPU affinity:
  - java MultiCastReceiver 230.0.0.1 dummy
  - java MultiCastSender 230.0.0.1 dummy 20000000
- Spuštění s použitím CPU affinity:
  - taskset -c 2 java MultiCastReceiver 230.0.0.1 dummy
  - taskset -c 4 java MultiCastSender 230.0.0.1 dummy 20000000



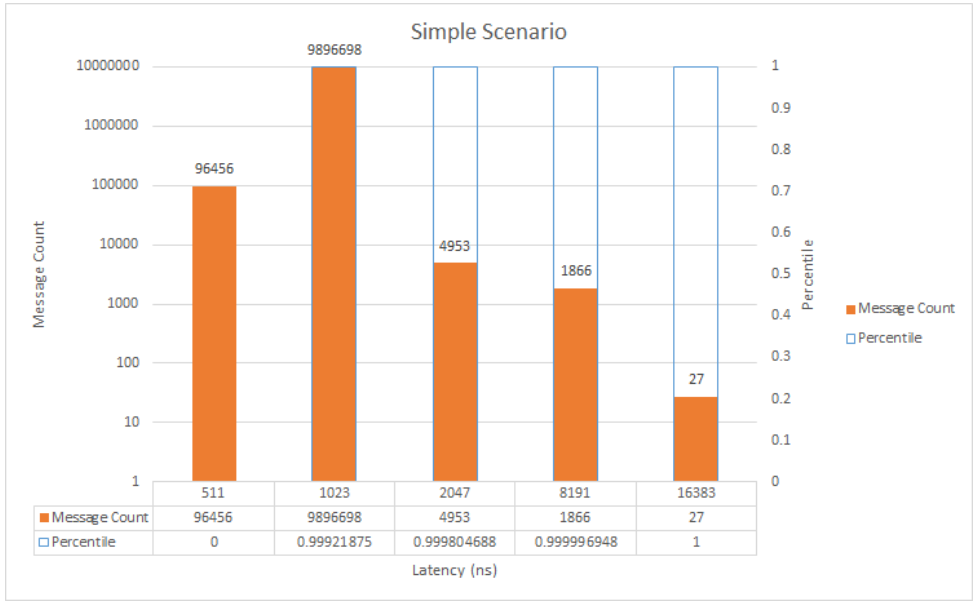
# DET Hub

- Univerzální obchodovací systém
- Umožňuje propojit libovolný počet systémů, které komunikují různými protokoly
- Jádro systému nezávislé na protokolu
- Uživatel nastavuje pravidla pro zpracování zpráv



# DET Hub - Výkon

- Žádné zámky
- Jádro systému nealokuje žádnou paměť na heapu
  - Nedochází k GC
  - Stabilní latence
- Speciální reprezentace zpráv v souvislých blocích paměti
- Unikátní persistenční vrstva



# Situace na trhu práce

- Česká Republika
  - poměrně málo rozšířený obor
  - málo firem, kde je možné získat potřebné zkušenosti před odchodem do zahraničí
- Zahraničí
  - velmi vysoká poptávka, především ve světových finančních centrech jako jsou Londýn, New York, Singapore, Tokyo a další
  - nadprůměrné finanční ohodnocení – např v Londýně se standardní plat contractora (75% trhu) pohybuje mezi 450 až 800 liber na den
- MFF poskytuje ideální soubor základních znalostí potřebných pro prosazení se v daném oboru

# Odkazy

- mechanical-sympathy.blogspot.com/
- lmax-exchange.github.io/disruptor/
- www.akka.io
- www.azulsystems.com
- www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

# Kontakt

- info@det-tech.com