***What do you understand by the*** *terms****real-time application (RTA)****,****latency & throughput?***

* ***Real-time application (aka RTA)***
  + *is an application where the content is pushed through “as it happens” within a specified time frame.*
  + *These time frames are defined as SLAs (Service Level Agreements).*
  + *For example, in a****S****traight-****T****hrough****P****rocessing (STP) solution, you have****real-time****trades flow between your front/middle office, and traders/stock exchange.*
* *Real-time systems can be further divided into****1)****Hard real-time &****2)****soft real-time*

*Hard real-time*

* *Action need to be done immediately in sec time*
* *when an action is performed at the wrong time will have possibly no or negative effect.*
* *It is not acceptable to say 90% of the time we hit the response time of 100ms. Only some systems have this requirement -> medical apps (e.g. pacemaker), defense systems, nuclear systems, avionics, etc.*

*Soft real-time*

* *is when an action is performed either too early or too late will still have a positive effect.*
* *If it had performed the task on time, it would have had greater value in terms of better customer experience, meeting the SLAs, etc. Soft real-time system can be a trading application with high through-put & low latency without any hard response time guarantees.*
* *No catastrophe happens when response times fail say 5% of time when 100K requests are sent. Most system fall into this category like financial applications (e.g. placing trades, matching trades, etc), event processing, telecom, etc.*

***Can JVM be used for real-time applications in the sense that it’s guaranteed to react within a certain amount of time?***

* *The answer is no for the standard JVMs,*
* *but the special JVMs that support “****Real-Time Specification for Java (RTSJ)****” extensions can process in hard real-time.*
* *Standard JVMs achieve “****soft real-time****” mainly due to automatic garbage collection and GC pauses associated with it.*
* *The RTSJ provides a subclass of* ***RTT******(i.e. Real-Time-Thread)*** *called* ***NoHeapRealtimeThread*** *(****NHRT****).*
* *Instances of this subclass are protected from GC induced pauses.*
* *NHRTs are NOT allowed to use the heap.*
* *NHRTs use the scoped memory and immortal memory features to allocate memory on a more predictable basis*

***Do you favor hard or soft real-time Java development guidelines in general?***

* *soft real-time is favored unless there is a specific need for hard real-time as soft real-time offers much better****developer productivity****&****application maintenance****.*

***Latency***

* *is the time required to perform some action or to produce some result?*
* *Latency is measured in units of time like seconds, milliseconds, micro seconds, nanoseconds, etc.*
* *What defines a “low” latency depends on the context – low latency over the internet might be 200ms whereas* ***low latency in a trading application (e.g. pricing or order matching engines) using FIX or custom protocols over TCP/IP might be 2µs****.*
* *Trading systems need to target 100 nanoseconds to 100 ms.*

***Throughput***

* *is the number of such actions executed or results produced per unit of time.*
* *This is measured in units of time like requests per second. The term “memory bandwidth” is sometimes used to specify the throughput of memory systems.*

***How do you achieve stream processing in JVM languages like Java, Scala, etc?***

* *You can build your own application with a Message Oriented Middleware (i.e. MOM) like Websphere MQ, ActiveMQ, RabbitMQ, Kafka, etc*
* *where you write code to receive events from topics in the broker (i.e. event streams), compute the results and then publish results back to the broker.*
* *Alternatively, and preferably you can use a stream processing framework like* ***Apache storm****,* ***Apache Flink****,* ***Apache Spark streaming****, etc to save time.*
* *The framework will do the heavy lifting by collecting data, delivering it to each processor, making sure they run in the right order, collecting results, scaling across nodes if the load is high, and handling failures by retrying.*

***What is the difference between the terms latency and response times?***

* *Latency is the time elapsed between when a request was sent to the server and when the first byte of the response has started to be received.*
* *Response time is the time elapsed between when a request was sent to the server and when the response has been fully received. In a web application the browser needs to load the assets like the DOM tree, images, CSS, and the JavaScript scripts.*

*So, the response time will always be >= latency. In other words,*

|  |
| --- |
| response time = latency + processing time (e.g. render the browser) |

*Low latency is a sum of many things, and two most important ones are*

*1.****Network Latency****, which is the time taken on the network to send/receive a message/event & 2.****Processing Latency****, which is the time taken by your application to act on a message/event.*

*If you are building a “trade order matching” engine in Java, the “****network latency****” is the time taken in say micro seconds to receive an order matching request to the engine from a client app plus the time taken for the client app to receive the first byte of the response message from the engine.*

*The “****processing latency****” is the time elapsed in micro or milliseconds for the engine to match the order and build the response to be sent back to the client app.*

***Is a latency of over 20ms considered fast or slow in HFT (High Frequency Trading) application?***

* ***Anything over 20ms will be considered slow****.*
* *The HFT trades are conducted using algorithms to buy, sell, and match huge volume of trades. These are ultra-low latency applications once used to be written in “C”, and now a day increasingly being written in Java.*

***What throughput will you be aiming for in HFT (High Frequency Trading) applications?***

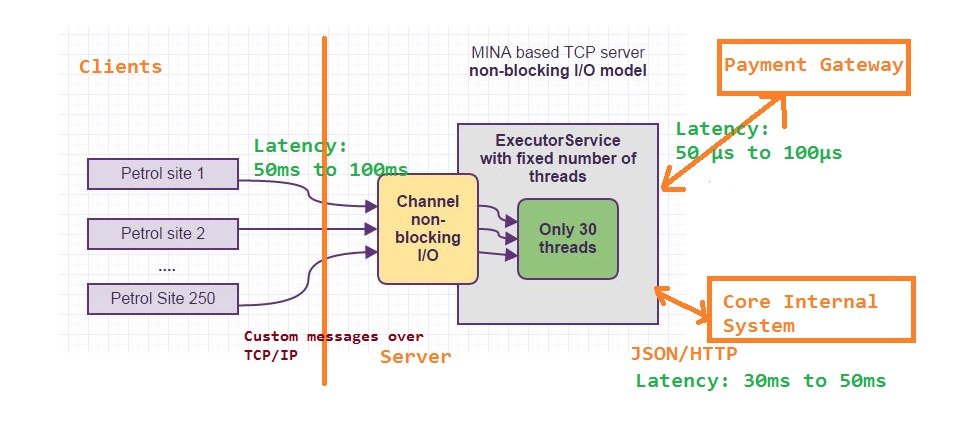
* *50k to 200k orders or transactions per second.*
* *You will have multiple servers to process the requests. The architecture needs to be****scalable****to cater for growing demands. You can learn more at*[*Scalability interview questions & answers*](https://www.java-success.com/scalability-interview-questions-answers/)

***What latency will you be targeting for your applications?***

* *It depends on the context of the application. For example,*

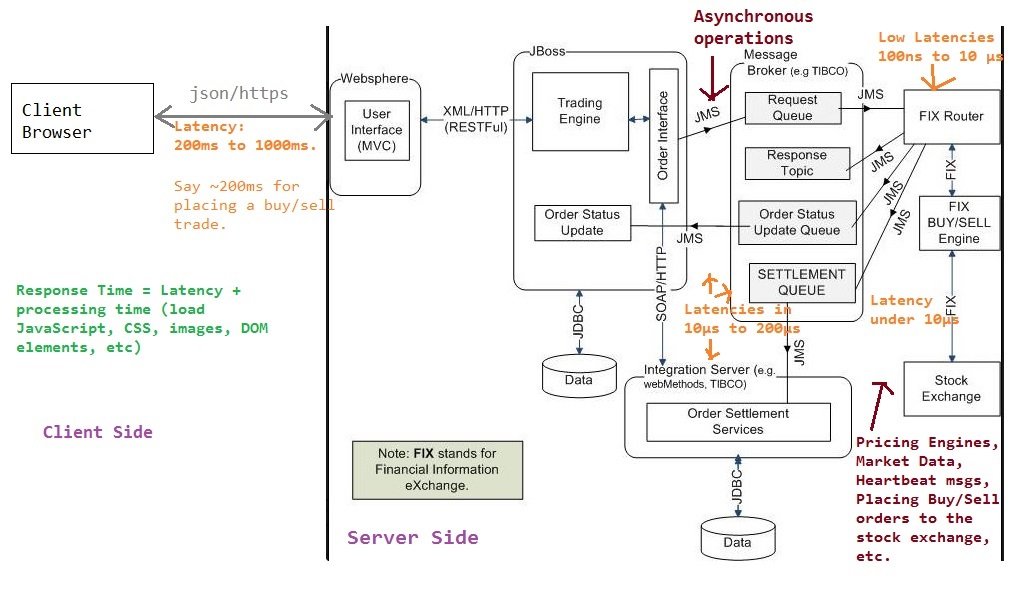
1. *Trading system placing buy/sell equity or FX orders to the market will target a latency of under 20ms.*
2. *A standard web application will target a latency of 200ms to 800ms.*
3. *A gaming application or a more complex web application will target a latency of 500ms to 1000ms.*

***Example 1: An EFTPOS system***

[](https://www.java-success.com/wp-content/uploads/2014/11/Pay-At-Petrol-Site-EFTPOS-Latencies.jpg)

EFTPOS Latencies example

***Example 2: An Online Trading System***

[](https://www.java-success.com/wp-content/uploads/2014/11/Latencies-trading-app-example.jpg)

Latency: industrial strength example

***How will you go about improving the latency for a more complex web site?***

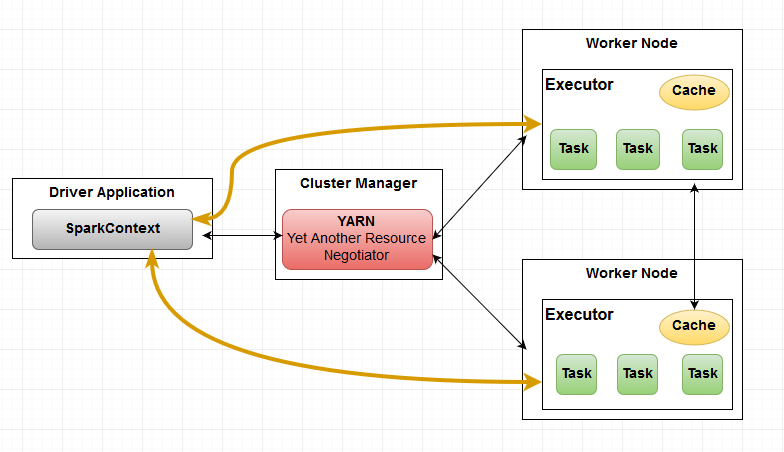
1. *Processing the requests asynchronously by submitting to a queue and getting the results later via a client pull or server push.*
2. *Reducing the complexity of the page by dividing the tasks with the view of better user experience. This also means smaller payloads transferred between the clients & the servers for better network latency.*
3. *Producing less garbage by violating the OO concepts by favoring primitive data types and applying the flyweight design pattern to improve reuse.*
4. *Profiling the application with the tools like VisualVM to identify and improve the bottlenecks in terms of CPU, memory usage, Garbage Collection pauses, etc.*
5. *Writing low latency Java*
   1. *Warm up your JVM. Bytecode starts off being interpreted for Hotspot and then gets compiled on the server after 10K observations. Tiered Compilation can be a good stop gap.*
   2. *Class loading is a sequential process that involves IO to disk. Make sure all the classes for your main transaction flows are loaded upfront and that they never get evicted from the perm generation.*
   3. *Follow the "Single Writer Principle" to avoid contention and the queueing effect implications of Little's Law, plus study Amdhal's Law for what can be parallel and is it worth it.*
   4. *Model your business domain and ensure all your algorithms are O(1) or at least O(log n). This is probably the biggest cause of performance issues in my experience. Make sure you have performance tests to cover the main cases.*
   5. *You need to understand the whole stack your code is executing on. This will involve OS tuning, selecting appropriate hardware, tuning systems software and device drivers for that hardware.*
   6. *Be realistic. If you need low-latency don't run on a hypervisor. Ensure you have sufficient cores for all threads that need to be in the runnable state.*
   7. *Cache misses are your biggest cost to performance. Use algorithms that are cache friendly and set affinity to processor cores either with taskset or numactl for a JVM or JNI for individual threads.*
   8. *Consider an alternative JVM like Zing from Azul with a pause-less garbage collector.*

***What do you understand by the terms real-time systems, latency, and scalability?***

***Real-time****and****low-latency****are distinctly separate subjects although often related. Real-time is about being more predictable than fast. Low latency systems need to be fast to meet SLAs (Service Level Acceptances) in sub milliseconds (e.g. micro seconds).Scalability means the ability of the system to handle growing demands by adding more CPUs, memory, nodes, servers, etc.*

***What are some of the considerations in writing low latency applications in Java?***

1. *Parallel computing via*
   1. *multi-threading*
   2. *Non-blocking I/O (E.g. MINA, Netty, Grizzly, etc)*
   3. *distributed systems (E.g. Apache Kafka, Apache Spark, etc) with****share nothing architectures****.* *Share-nothing architecture is where applications will be running in parallel on 100+ nodes with its dedicated CPU, memory, I/O, etc.*
2. *Streaming APIs like Apache Spark streaming, Apache Storm, StAX (i.e.****S****treaming****A****PI for****X****ML) to process data in real-time or near real time.*
3. *Writing concurrent programs with Java multi-threading features such as executors, futures, completable futures****,*** ***fork/join****, concurrent data structures, etc.*
4. *Understanding the Java memory model & tuning memory & garbage collection in Java.*
5. *Using event based and non-blocking paradigms. For example, using frameworks like Apache****MINA****,****Netty****, Grizzly, and****Akka****.*
6. *MINA & Netty are lower level frameworks than Akka and have NIO (New Java IO) as its core**. NIO is an event driven non blocking paradigm.*
7. *Even though you need to have a good handle on writing concurrent programs in Java & interviewers like to quiz/test you on it, favor a framework like Akka as writing complex concurrent programs is not a trivial task, and you need to deal with threads, locks, race conditions & debugging. Writing concurrent programs without frameworks can be error-prone and can lead to code that is difficult to read, test, and maintain.*



SparkContext with Executors executing tasks

Q9. What is an **actor model** in Akka toolset, which is also known as the **reactor design pattern**?  
A9. **“actor model”** is a design pattern for writing concurrent and scalable code that runs on distributed systems. This is an **event driven** (i.e. message passing) model that involves sending & receiving events among actors.

**1)** Instead of invoking an object directly, you construct a message and send it to a destination object called an **actor**.

**2)** Each thread is an actor with a specific job to do. The actor engine stores the message in a queue.

**3)** When a thread becomes available, the actor engine running the actor delivers that message to its destination actor object.

**4)** When the actor completes its task, it sends a message back to the originating object, which is also considered an actor.

**5)** You can orchestrate which messages get passed to which actors under what conditions.

The **akka-camel** module allows “Untyped Actors” to receive and send messages over a great variety of protocols such as HTTP, SOAP, TCP, FTP, SMTP or JMS and APIs such as java & Scala .

*Low latency techniques for interview*

1. *Parallel computing via*
   1. *multi-threading*
   2. *Non-blocking I/O (E.g. MINA, Netty, Grizzly, etc) .*
   3. *distributed systems*
   4. *Share-nothing architecture is where applications will be running in parallel on 100+ nodes with its dedicated CPU, memory, I/O, etc.*
2. *Use Streaming APIs*
3. *Use fork/join (java 8)*
4. *Lamda streaming*
5. *Understanding the Java memory model & tuning memory & garbage collection*
6. *Use event based and non-blocking paradigms. For example, using frameworks like Apache****MINA****,****Netty****,* ***Grizzly, and******Akka****.*
7. *Use NIO. NIO is an event driven non-blocking paradigm.*
   1. *Channels &Buffers*

*In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer or written from a buffer to a channel.*

* 1. *Selectors (Chat application)*

*Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.*

1. *Use design pattern like* ***reactor design pattern***