## Random Points

One of our project uses ‘CPD’ to process high volume real time data. CPD stands for Collector, Parser and Distributor. The design behind CPD is asynchronous processing using intermediate queues. Application need to get data from various sources and provide it to end user. This task is broken into three tasks - Task of collecting data in raw format (Collector), parsing it format application can understand (Parser) and sending it to user (Distributor), where output of a task becomes input to next task which is shared using queues.

*Input -> C* – Q – *P* – Q – *D -> Output*

Major concept is to have separate thread for each tasks, each thread gets input from shared queue and puts its output in another shared queue or relay it to end user or store in database. Output of a task becomes input to another task. This type of architecture is very useful in certain scenarios. It provides various advantages –

1. Resources are limited in your system, it provides alternative while utilizing resource available.
2. It provides various decisions can be taken at different step, like for Quodd CPD, we may want to ignore messages which are expired i.e. more updated data is already available.
3. It utilizes your system resources, if you process different requests in separate threads most of the time CPU is taking in context switching, where in this case when one of your thread gets CPU time it can utilize that time to process more requests as it can process bulk of data available in queue at that time
4. It do add latency in your system so may not suitable for all type of applications, take your decision on based on related factors like type of request, delay expected, number of CPU cores available in your system as it is more beneficial when you can provide dedicated cores to each thread or close to it, etc.
5. CPU utilization is good in such scenarios as these simulate SIMD/MIMD at software/application level. CPU is kept busy or utilized when your process/thread does some work, small work never increase your CPU utilization.
6. Provide the ability to divide big task into smaller task which improves design of application. Addition of new task can be done like intelligent filters, just by changing queues tasks operate on.

Note - This architecture processes around 1TB real time data in 7 hours on daily basis at Quodd server machine.

“The 64k Connection Myth”

It’s a common misconception that you can only accept 64,000 connections per IP address and the only way around it is to add more IPs. This is absolutely false.

The misconception begins with the premise that there are only so many ephemeral ports per IP.

The truth is that the limit is based on the IP pair, or said another way, **the client and server IPs together**. A single client IP can connect to a server IP 64,000 time and so can another client IP.

**Abstraction –**

Abstraction plays very great role in application design, development and execution. Apply abstraction properly at design phase. Abstraction provides various advantages –

1. Allow different components to be developed in parallel
2. Allows implementation to change without effecting other components of application
3. It provides reusability and readability to your code
4. It simplifies design
5. Other parts of your code can depend on abstract interface/classes and not to worry about implementation changes. This is called Dependency Inversion Principle (DIP) – Depend upon Abstraction. Do not depend on concretions.

Apply these simple rules to find components which are candidate for abstraction –

1. *Components which are expected to change in near future*
2. *Components which are expected to work differently on different data/scenarios but operation output remains same*
3. *Components for which implementation is unknown*

Take an example of CPD above –

What components are eligible for abstraction?

I see 4 components for abstraction –

Collector (IDataCollector)– There could be different sources from where data need to be collected, each having different protocol to send data.

Why IDataCollector? Why not ICollector? Because ICollector is very abstract name, and there may be some other Collector in application like for billing and other services etc, which are not supposed to have similar functionality.

*Class/Interface name should describe what it is intended to do*, so why not IStockDataCollector? Name should explain what it is intended to do in a given context, Application context is Stock Market, so it can be understood easily, however implementing classes could be named as OptionDataCollector, EquityDataCollector etc.

Parser (IDataParser) – There is expected to be a parallel hierarchy between Collector and Parser, for each type of Collector there should be a parser to parse data.

Distributor (IRecordDistributor – output from parser is record not raw data) – There could be different destinations to which output to be sent, over the wire, to database, to mobile application, so there could be different distributors.

Queue – This is a candidate of Data abstraction, there could be different queue implementation like shared FIFO local queue, Mapped queue, distributed queue (Active MQ) etc,

MYSQL Cluster – Varchar/Text –

MYSQL Cluster (NDB Storage Engine)

Only the first 255 bytes of BLOB and TEXT columns are stored in the main table with the rest stored in a hidden table. This means that what appears to your application as a single read is actually executed as two reads. For this reason, the best performance can be achieved using the VARBINARY and VARCHAR types instead. Note that you will still need to use BLOB or TEXT columns if the overall size of a row would otherwise exceed 8,052 bytes.

Java - Iterating over LinkedList -

Indexing into a LinkedList<T> is not a constant-time operation. Hence the loop

below takes time quadratic in the size of the list lst if lst is a LinkedList<T>, and should

not be used:

int size = lst.size();

for (int i=0; i<size; i++){

System.out.println(lst.get(i));

}

Instead, use the enhanced for statement to iterate over the elements. It implicitly uses the collection’s

iterator, so the traversal takes linear time:

for (T x : lst)

System.out.println(x);

***API - collection for primitive***

http://trove.starlight-systems.com/

The collection classes can store only reference type data, so a value of primitive type such as int,

double, . . . must be wrapped as an Integer, Double, . . . object before it can be stored or

used as a key in a collection. This takes time and space and may be unacceptable in some application.

If you need to use collections that have primitive type elements or keys, consider using the Trove

library, which provides special-case collections such as hash set of int and so on. As a result

it is faster and uses less memory than the general Java collection classes.

1. Open Closed Principle (OCP)

This principle states classes/components should be open for extension but not for modification. By using proper abstraction and inheritance (polymorphism) you can avoid changes in your existing working code, thus your code is open for extension but not for modification. KISS principle also helps you achieve better of OCP, think of a very big method in a class, how extension to that method can be provided? If your method is too big, extending class need to replicate all of the implementation even they want to change small sub behavior of that class/method.

1. Don’t repeat yourself (DRY)

You should not repeat code in your application. Figure out things which are common and place it in a single location. This makes your code more readable, reusable, maintainable and less error prone.

1. Single Responsibility principle (SRP)

Every object in your system should have a single responsibility and all services object provides should be fulfilling given responsibility. It is a bit different from KISS Stupid clause, as Stupid only states object should do what it is intended to do and should not have knowledge about external world, where SRP forces your object should not be doing more than one logical thing. This makes your code more manageable as if there is some requirement change or bug in your code; you know the exact location to make change to.

1. The Liskov Substitution principle (LSP)

Derived classes should be substitutable for their base classes. This principle is about well-designed inheritance. When you inherit from a base class, you must be able to substitute your subclass for that base class. If there are n methods in your base class, sub class should have implementation for those n methods with proper meaning in implementation. A simple example is Playing Board and 3DBoard, at conceptual level both are board, but most of the methods in Board class operate on x, y where in 3DBoard it needs x, y, z. So 3DBoard is not substitutable for Board and it should not extend Board, however it is using its behavior that can be done using delegation, composition or aggregation, but it is not a candidate of inheritance.

1. Kiss (Keep It Simple Stupid)

You can read it as “Keep it simple and stupid”. In programming world it states keep your code as simple and stupid. Simple - Never complex your code, your code should be easy to understand, break your code in to different methods if any method became too large, refactor it to level where it is easy to understand, methods/classes/variables should be self-explanatory. Stupid – means your code should be such stupid that it know only about what it is intended to do, should have minimal knowledge of external world or not at all.

Keeping it simple make your code easily understandable and manageable. Keeping it stupid makes it less error prone and manageable.