Umeå Universitet
Department of Computing Science

Distributed Systems 7.5p
5DV186

Assignment 1 - Performance Analysis

Date of submission
2017-11-16

Author:
Lorenz Gerber (dv15lgr@cs.umu.se)

Supervisors:
Per-Olov Östberg,
Jonny Pettersson
Adam Dahlgren Lindström

1 Introduction

The aim of this assignment was to implement serial and parallel video streaming clients by making use of a provided mock distributed stream server setup. The low-level communication details were abstracted by a high-level client/server communication stub. Further, an interface description in Java was given that had to be followed for implementing the clients. The implementation language was Java 8.

2 Interpretation of Specifications

The given specifications were in many points rather loose, therefore follows below a short account on how they were interpreted for the presented implementation.

2.1 General Code Structure

The given interface 'FrameAccessor' was left unchanged except that the Factory was extracted into a separate Interface (It was not clear to the author how an internal class should serve as a factory method to it's enclosing class). The interfaces were renamed to 'IFrameAcessor' and 'IFactory' respectively to allow 'clean' names for the actual classes.

The 'FrameAccessor' class was generic for serial and parallel stream access. Hence, the only difference between serial and parallel operatoin is the number of services handed to the 'FrameAccessor' Class.

2.2 Serial vs Parallel

For the current implementation, 'serial' was understood as a client that accesses one 'service' and requests the frames/blocks sequentially, single-threaded. Other ways of interpretation would have been to access one service through one connection using several threads or accessing one service through several connections using several threads.

For 'parallel' streaming, the four given services are accessed concurrently using four threads. The threads are kept alive during the whole session using a thread pool, while single concurrent jobs are generated as runnables at the level of block requests.

As mentioned before, 'serial' and 'parallel' stream access differ only in the number of services provided on the command line. Hence, there is only one client program that takes the command line arguments:

```
timeout username service [service service service].
```

2.3 Package drops and Timeouts

For the given application, 'streaming video client', it was decided that both package drops and timeouts are acceptable without taking special measures such as resend requests etc. This decision was also supported by the practicality that the stream service was very slow which would have made it even more difficult to collect enough data for performance statistics.

3 Performance Evaluations

The obtained throughput for the application was slow both serial and parallel. It was not clear to the author whether this was due to some mistakes in his implementation or whether there was a problem with the distributed service mocks. Since many other students reported

similar experiences, no further time was invested to research the deeper cause of the problem. However, the slow speed affected the type and amount of performance measurements that could be collected for obtaining meaningful information. To keep measurement times reasonable, a smallest frame stream (stream 7) and only 10 frames were used for performance evaluations.

3.1 Implementation of Metrics

As given from the interface specification, a 'PerformanceStatistics' object contained `get` methods for the four requested metrics. The `PerformanceStatistics` class is an inner class of the `FrameAccessor` class which contains a `get` method for a `PerformanceStatistics` object. Below follows a short account on the chosen implementation of the requested metrics.

- packet drop rate - this metric was measured per service. Before each block request, a service specific block counter was increased. On a timeout exception, another counter for dropped blocks (situated in the exception handling routine) was increased. The percentage of dropped block is calculated as `100/totalBlockCount*droppedBlockCount`.
- packet latency - this metric was collected for every successful received block calulated from milliseconds system timestamps immediately before a block request and directly after returning from the request. Dropped/timeout blocks are not accounted for.
- frame throughput (frames per second) - The total time was measured from instantiation of the `FrameAccessor` to the shutdown of it's thread-pool. Frames are counted per request to the `getFrame` method of the `FrameAccessor` object.
- bandwidth utilization (bits per second) - Each pixel contained 3 byte (rgb), and a block contained 16*16 pixel, hence 768 Bytes or 6'144 bits. Only successful transferred blocks, obtained from the service-wise metrics were included. The time base was the same as before: from instantiation to thread-pool shutdown.

Additional to the requested metrics, all latency measurements were exported per stream as raw data to allow a more rigorous statistical analysis.

3.2 Performance Results and Discussion

As data throughput was very low, stream 7 was chosen as it has the smallest frame size. Further, only 10 frames were consumed for each measurement. This resulted in run times of about 3.5 to 4.5 minutes.

Characterization of Individual Streams

To get an idea of the variation between all provided services, they were all accessed serial. The obtained latency raw data was then exported and plotted as tukey box plots (figure **??**. It can be seen that the median was similar for all services, but number and magnitude of outliers varied. Median values ranged from 151 to 158. It was noted that the cutoff corresponded to the chosen timeout value (1'000ms in this case).

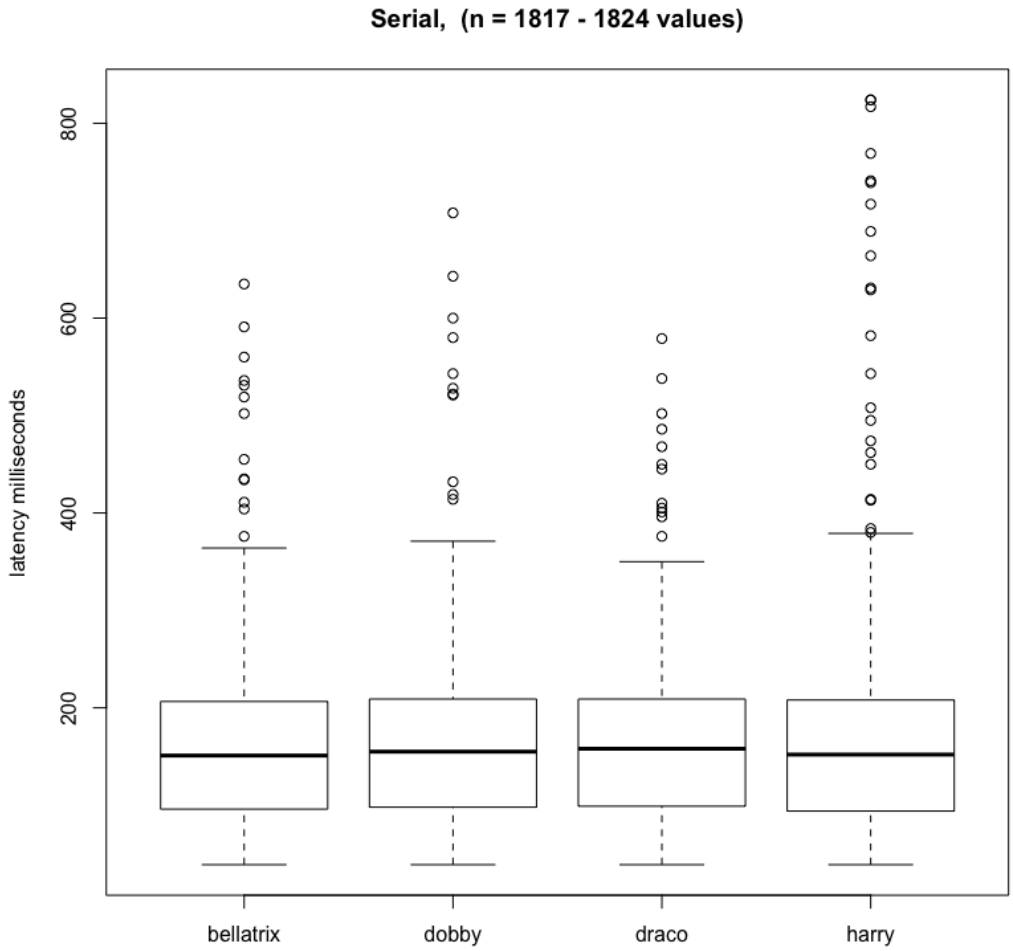

Figure 1: *Boxplot of latency values for each service, serially accessed, timeout 1'000ms (median, 1st and 3rd quartile, outliers if more than 1.5 away from the inter-quartile range)*

Table 1 *Comparing application wide performance metrics. The first four columns show serial client data where 10 frames from stream 7 were consumed. The rightmost column shows metrics for threaded parallel streaming with all four services.*

	draco	dobby	harry	bellatrix	parallel - all services
frames per second	0.03	0.03	0.03	0.03	0.04
bits per second	28'652	29'166	28'966	29'804	45'969

Table 2 *Comparing the stream individual metric 'drop rate'. Note that the serial measurements are based on 10 frames from stream 7 while for the parallel measurements each service only accounts for 1/4 of the total streamed data.*

drop rate in %	serial	parallel
harry	5.21	8.75
dobby	5.1	6.04
bellatrix	5	7.92
draco	5.36	7.29

Comparison Serial vs Parallel Clients

First application wide metrics were composed in table 1. With the highest bits-per-second throughput serial stream (bellatrix) as reference, the throughput increased by about 50% when the number of threads/clients was increased by 400% (from 1 to 4 threads/services). The 'frames per second' gives an idea of how slow the implementation was.

In table 2, the drop rates per serial service and for all in parallel was composed. While both serial and parallel experiments were run with the same stream timeout value, the parallel measurements exposed a significantly higher drop rate. It is not clear to the author whether this is a simulated feature of the distributed mocks or whether this is a problem of the parallel implementation. The result seems strange, as the time accounting is within the runnable where 'parallel overhead' operations should not affect. More in-depth code profiling would probably help to find the cause of this issue.

The latency values aggregated in the java program are simple means. From the boxplots in figure **??** it was clear that the distribution of latencies would not be well represented by means, hence for a better comparison, all latency values were exported as raw data and box plotted in figure **??**. Note that the y-scale is not the same in the serial and the parallel plot. Latency values were much higher for parallel stream access. Median values ranged from 209 for 'dobby' to 409 for 'draco' service. Interestingly, cutoff values here do not correspond to the set timeout of 1'000 ms. Hence, it can be speculated that a lot of time is wasted in the client code related to threading. In the current implementation, a thread pool is used. The runnables, however are provided per block access. Probably, parallel overhead could be reduced when for each frame runnable sets with multiple blocks would be composed and submitted to the thread pool. On the other hand, also the single threaded serial stream access uses runnables on a block access level and still performs much better.

3.3 Further Discussion

Below follows a short discussion on further topics requested in the assignment specifications.

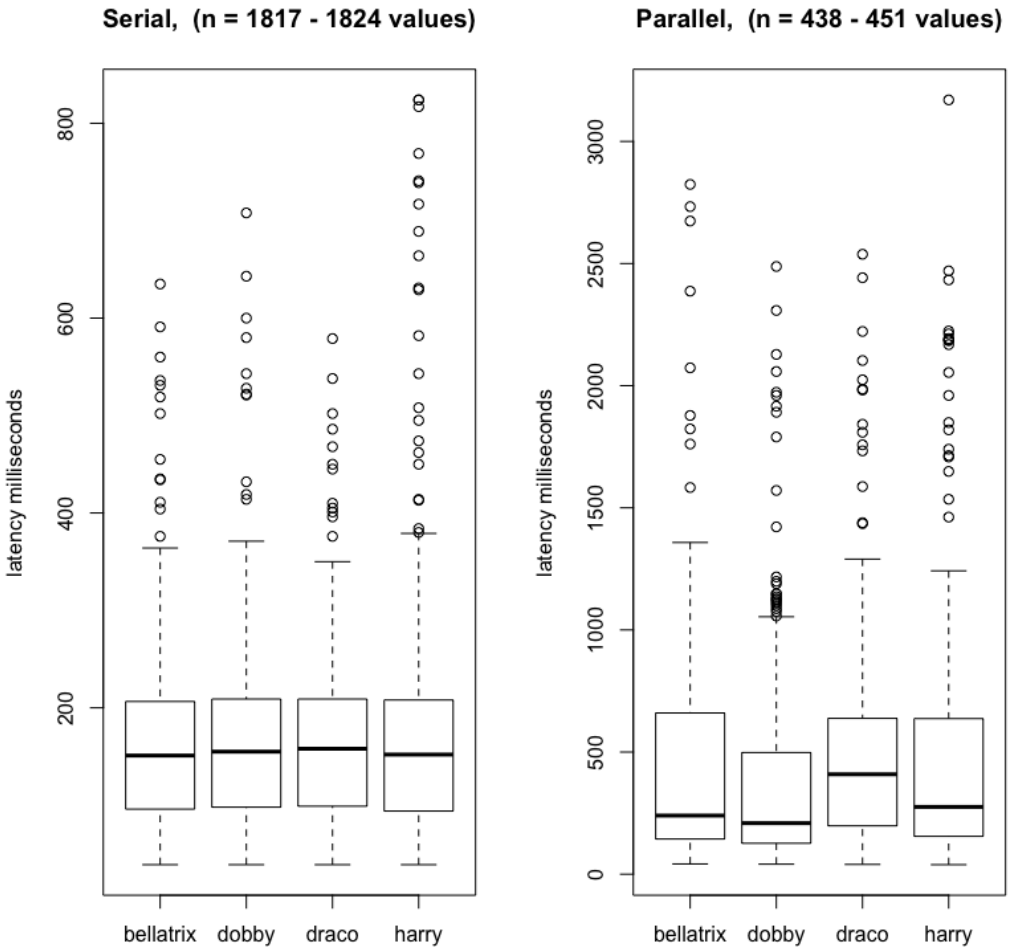

Figure 2: *Boxplots of latency values for each service serially accessed on the left and extracted for each service during parallel streaming access on the right (median, 1st and 3rd quartile, outliers if more than 1.5 away from the inter-quartile range)*

Effect of Packet Drop on Latency and Throughput

The effect of packet drop on latency depends on how it is interpreted. Either for a dropped package the cut-off time is used or the value is discarded. If measures to remedy packet drop, such as 'packet resends' are implemented, the total time to successfully obtain a packet should be used. Then packet drops have a high impact on latency. In the current implementation, no measures were taken in the case of packet drop, hence it was decided to discard latency data points from dropped/timeout blocks.

For throughput measurements, obviously, dropped packets should not be included. Only successful received packets should be included in the metric.

Effect of Compression Techniques

compression techniques have no influence on the actual throughput in bits per second. However, as the same amount of bits contains 'more data', the frame rate would increase. As each package contains now a 'larger image area', dropped packets will have a much higher influence on the experienced quality of the image. Probably, a compressed stream will need packet resends and a certain amount of buffering to achieve a reasonable image quality.