BAB IV

HASIL PENELITIAN DAN PEMBAHASAN

1.1 Lingkungan Pengujian

Percobaan dilakukan dengan menggunakan sebuah *notebook* dengan spesifikasi sebagai berikut:

Operating System	Window 10 Pro 64 bit build 14393
Processor	I5-2410M CPU @ 2.30 GHZ (4 CPUs)
Memory (RAM)	8 GB
Memory (Heap Max)	1.77 GB
Java Version	1.8.0_111-b14 (64 bit)

Adapun percobaan akan dilakukan dengan mengunakan dua buah modul, yaitu:

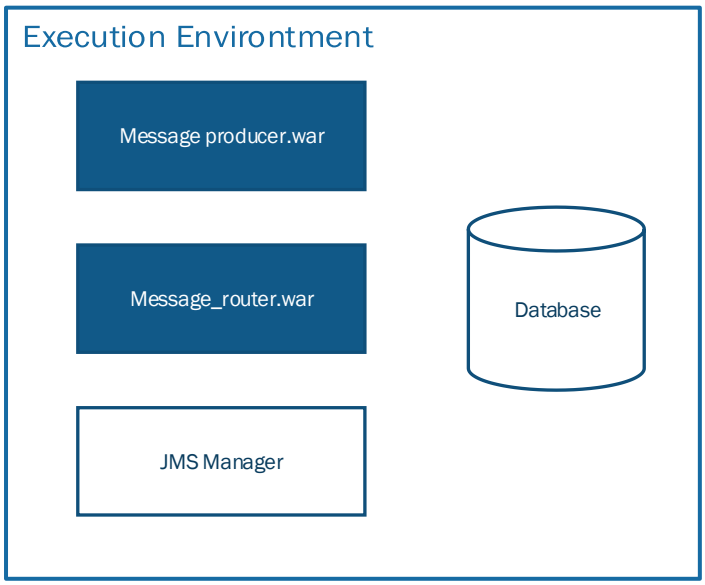
1. *Message producer*

   Akan membaca sejumlah *string* yang telah disiapkan di dalam file, kemudian secara acak akan mengirimkan *message* tersebut dalam bentuk *object* secara *streaming* via TCP ke JMS *queue*.

2. *Message router*

   Akan mengambil *message* dari JMS queue kemudian akan diolah dengan menggunakan salah satu dari kerangka kerja yang ada. Adapun pemrosesan yang terjadi adalah *mapping* dari *rule table* di *database* (*content manipulation*).

Kedua modul ini akan dijalankan dalam sebuah lingkungan pengujian yang sama, *deployment diagram* dari sistem ini dapat dilihat pada gambar di bawah:

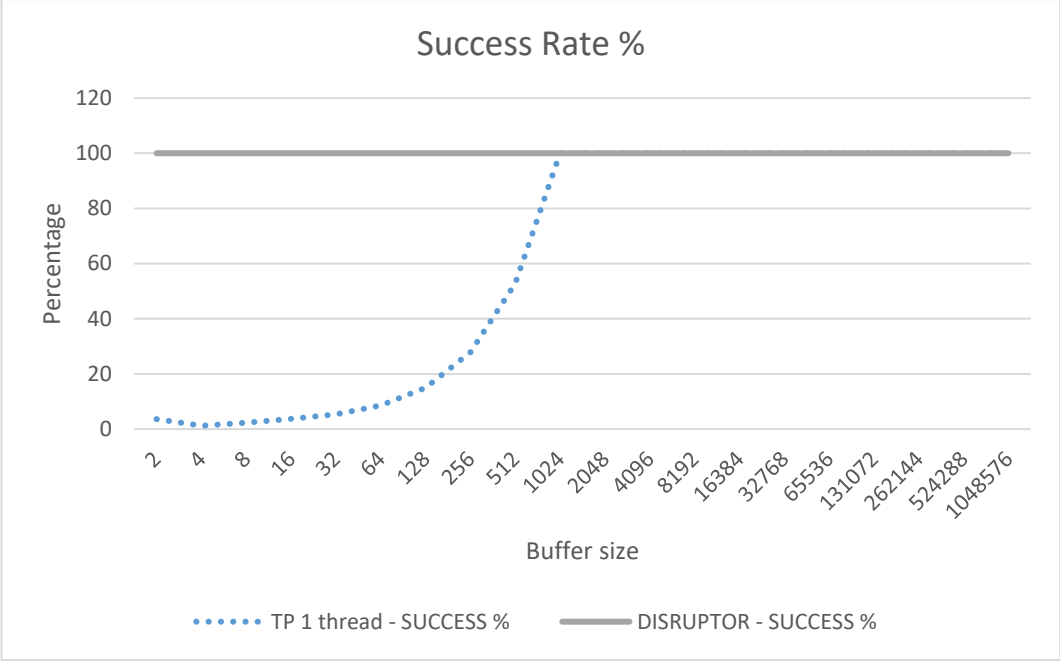


Gambar 4.1 Deployment Diagram

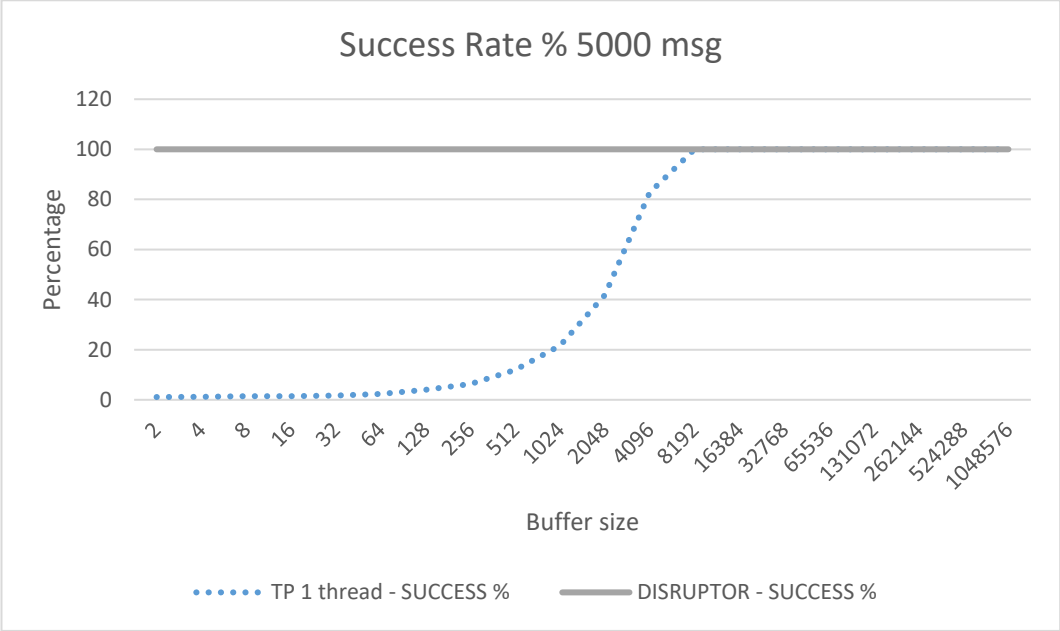
1.2 Hasil pengujian dan analisis data

Pengujian dilakukan dengan menjalankan komponen *message* router hingga dalam posisi *stand by*, lalu dilanjutkan dengan menjalankan komponen *message producer* dengan setting berapa *message* yang akan diproduksi. Setting pada *message* router adalah jenis *framework* yang digunakan dan *buffer size* nya. Dalam sekali eksekusi, sistem mampu mencatat data statistik untuk satu nilai *buffer size*, sehingga pengujian akan diulang sebanyak jumlah *buffer size* yang akan diuji yaitu 10 kali per jumlah *message* yang akan diuji, dalam hal ini jumlah *message* yang diuji adalah 1000, 5000, 10000, 30000, 60000, 120000, sehingga pengulangan dilakukan sebanyak 10 x 6 x 2 (dua buah *framework*).

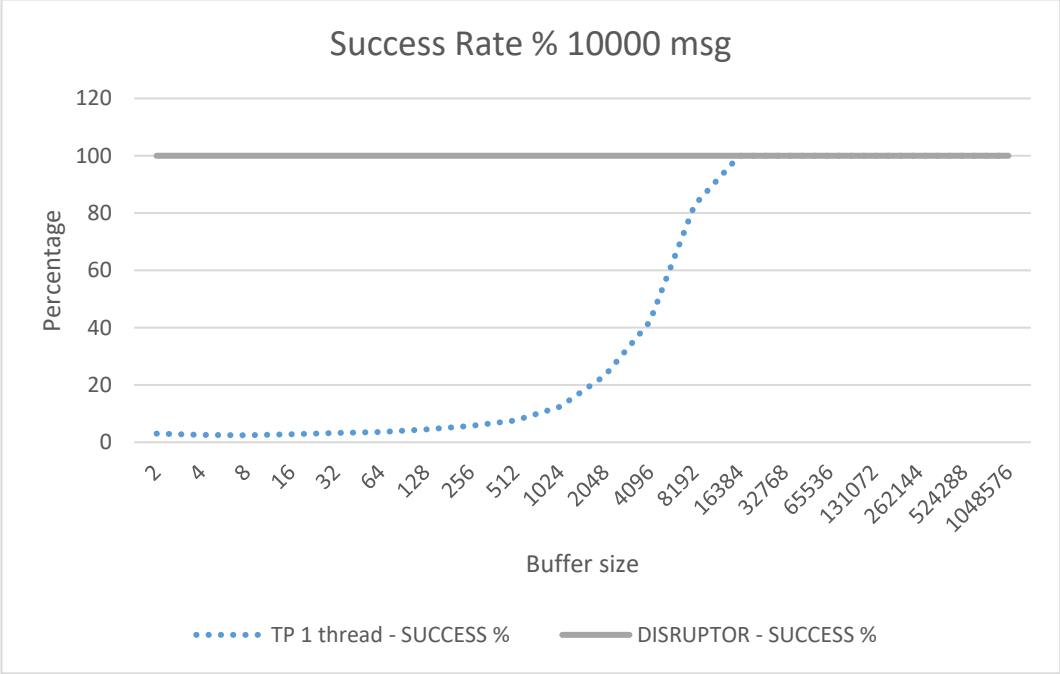
1.2.1 Success rate



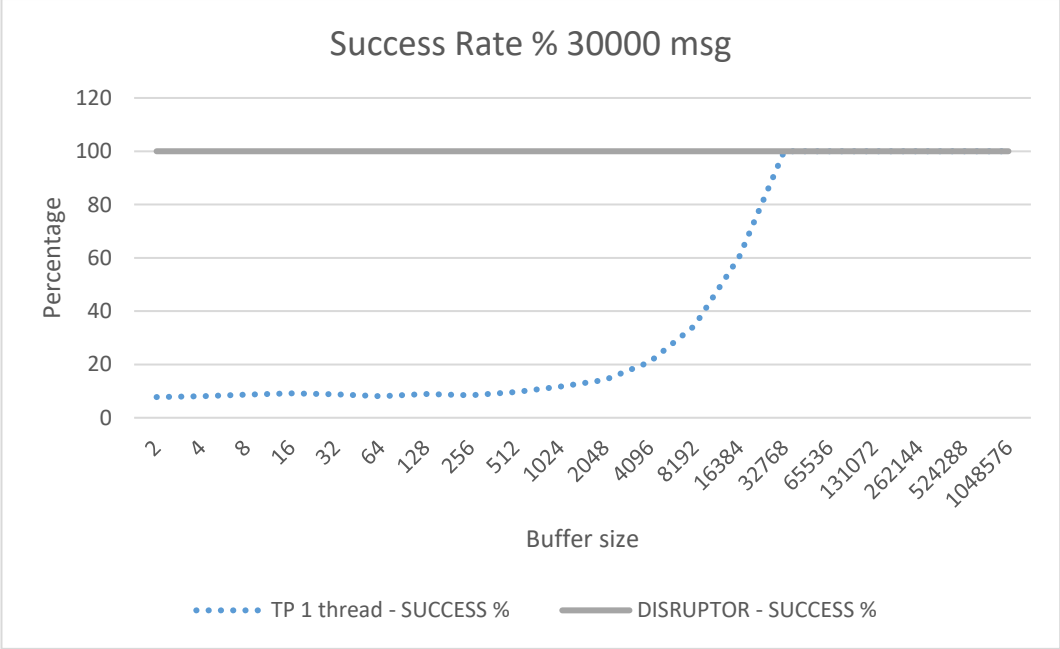
Gambar 4.2 *Success Rate* untuk 1000 *messages*



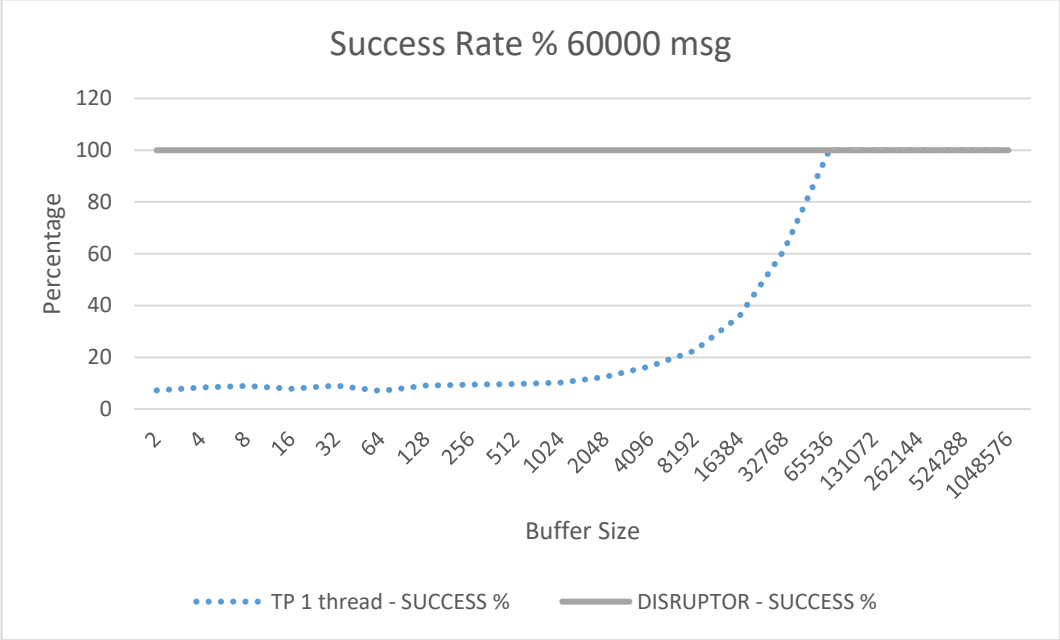
Gambar 4.3 *Success Rate* untuk 5000 *messages*



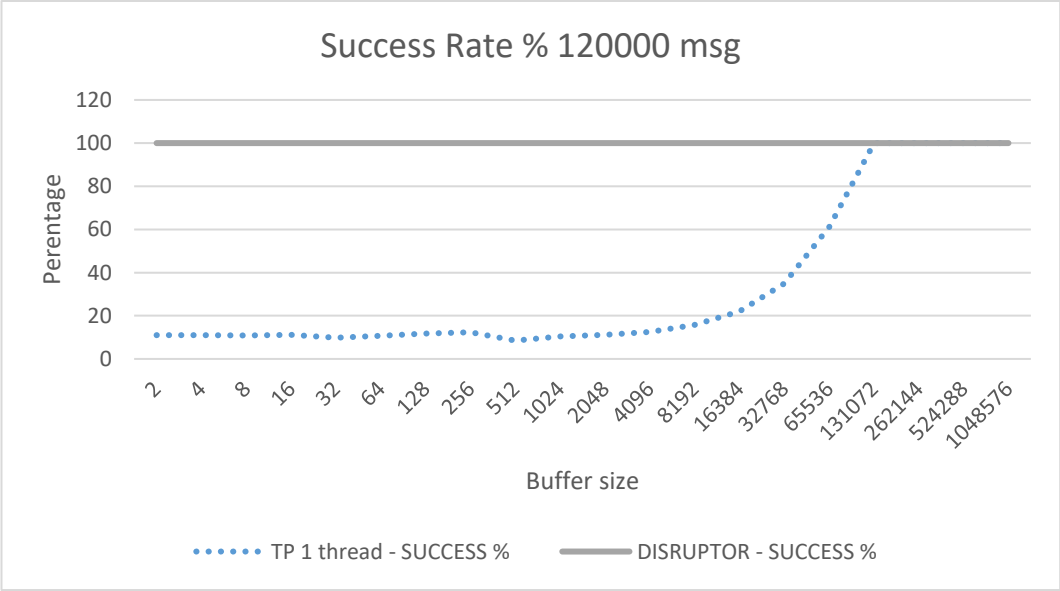
Gambar 4.4 *Success Rate* untuk 10000 *message*s



Gambar 4.5 *Success Rate* untuk 30000 *message*s



Gambar 4.6 *Success Rate* untuk 60000 *message*s

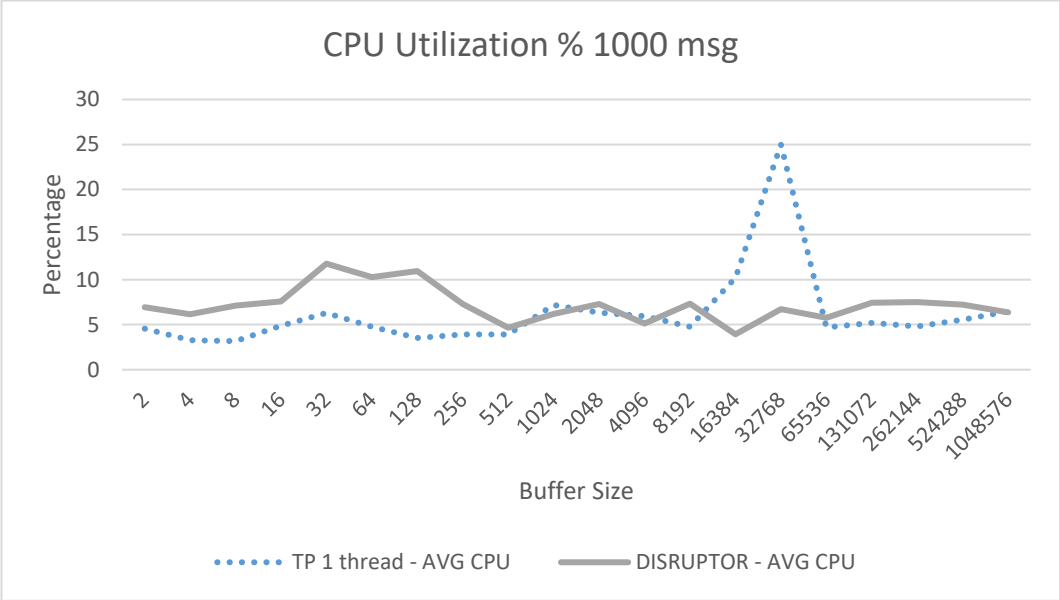


Gambar 4.7 *Success Rate* untuk 120000 *message*s

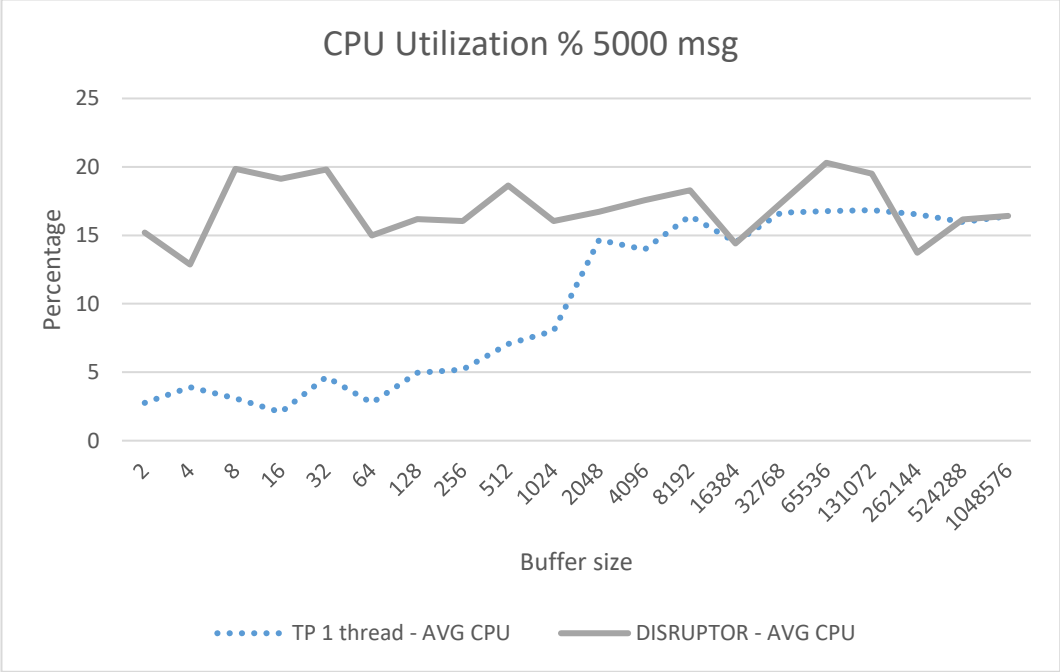
Untuk *message* dengan jumlah sedikit maupun banyak, LMAX Disruptor memiliki konsistensi rate positif 100%, hal ini berbanding terbalik dengan Threadpool, *success rate* yang dimiliki selalu dimulai dengan nilai yang rendah

(0%) dan perlahan meningkat seiring dengan bertambahnya *buffer size*. Pada satu titik, Threadpool akan memiliki *success rate* yang sama dengan LMAX Disruptor, titik ini berbeda untuk setiap jumlah *message* yang diproses, dengan jumlah *message* yang semakin besar, maka *buffer size* yang dibutuhkan semakin besar pula.

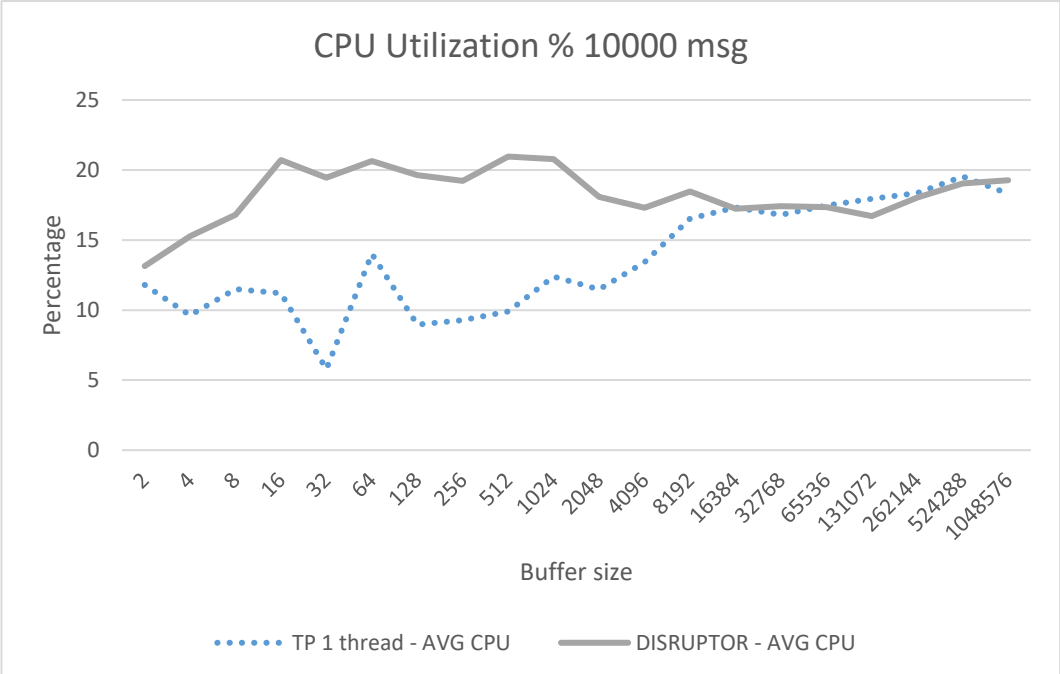
1.2.2 CPU *utilization*



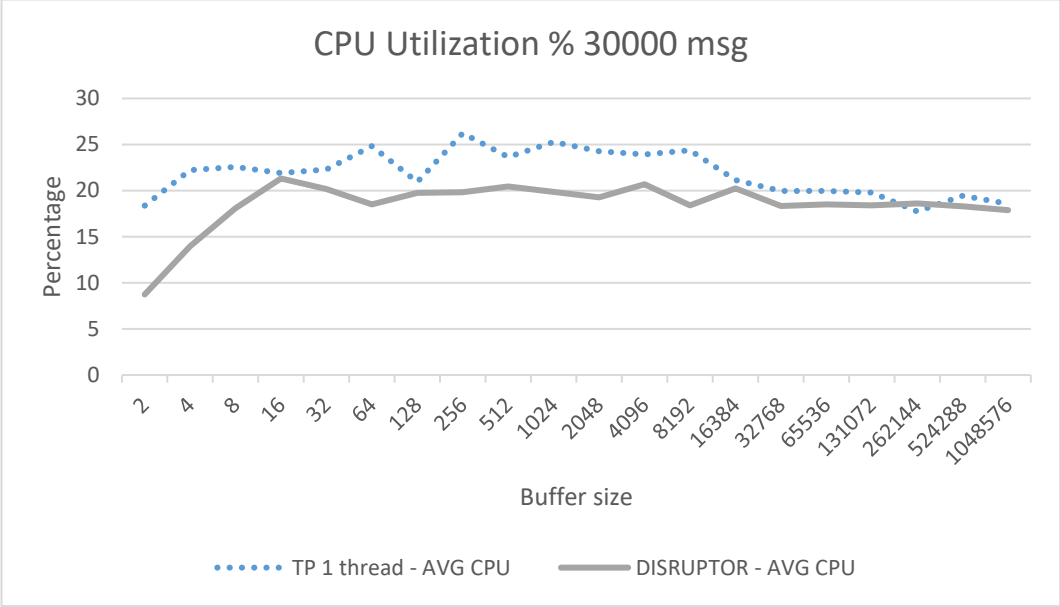
Gambar 4.8 CPU *utilization* untuk 1000 *message*s



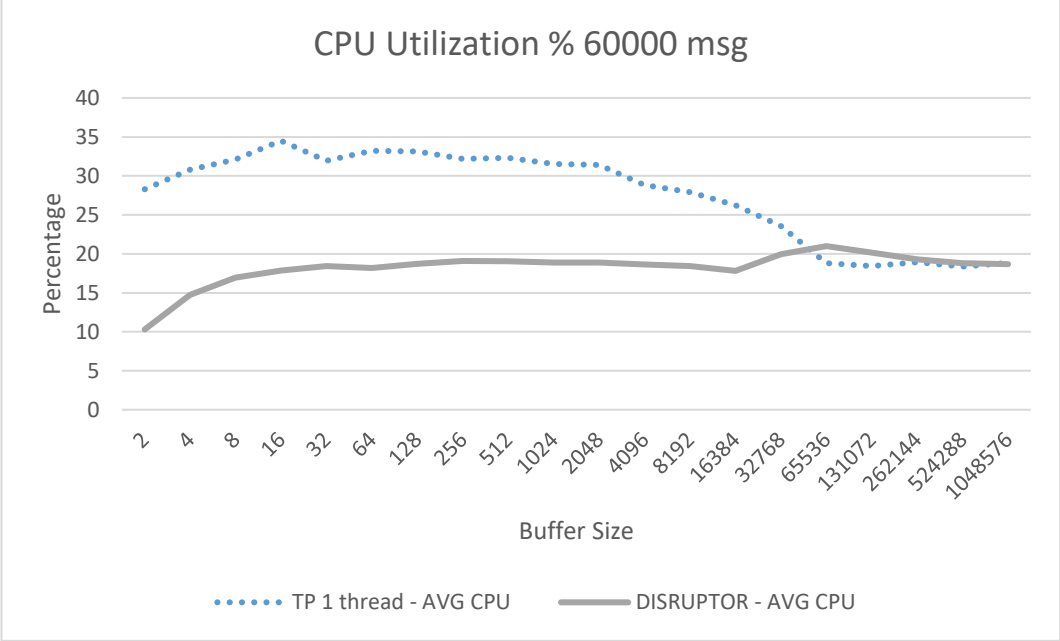
Gambar 4.9 CPU *utilization* untuk 5000 *messages*



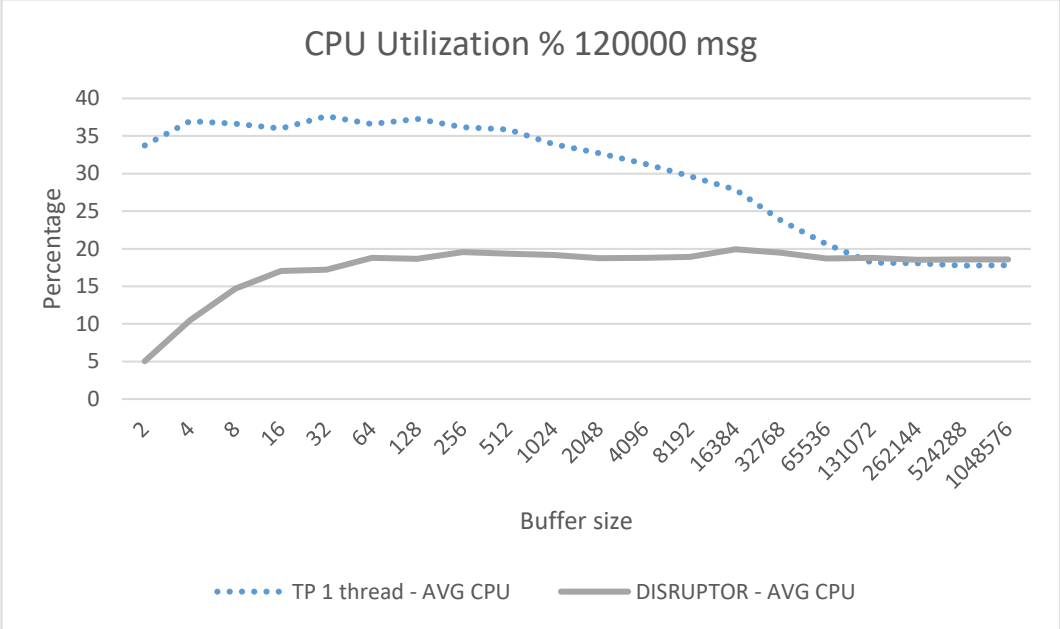
Gambar 4.10 CPU *utilization* untuk 10000 *messages*



Gambar 4.11 CPU *utilization* untuk 30000 *message*s



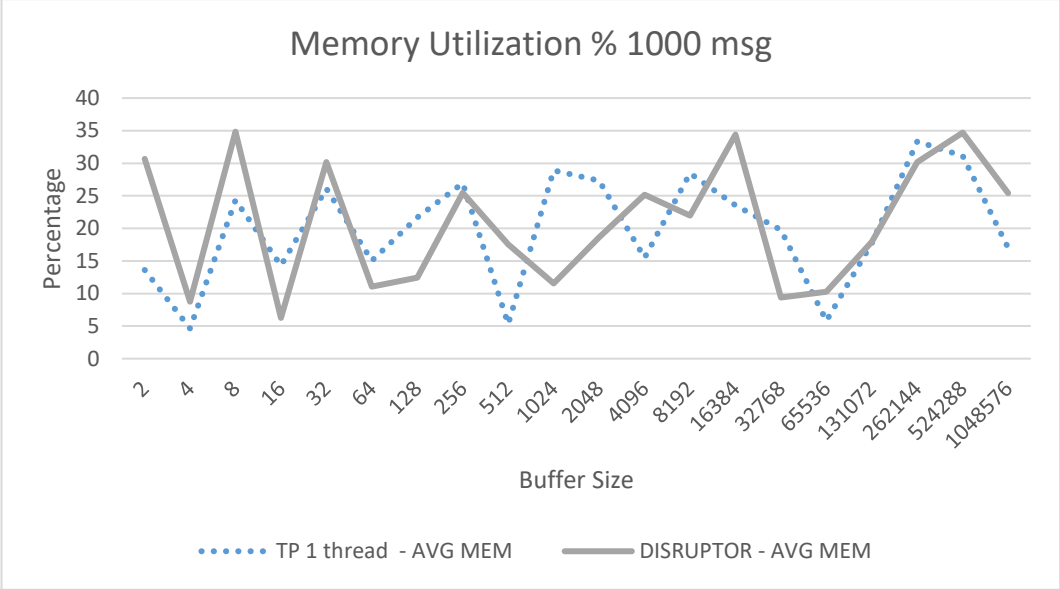
Gambar 4.12 CPU *utilization* untuk 60000 *message*s



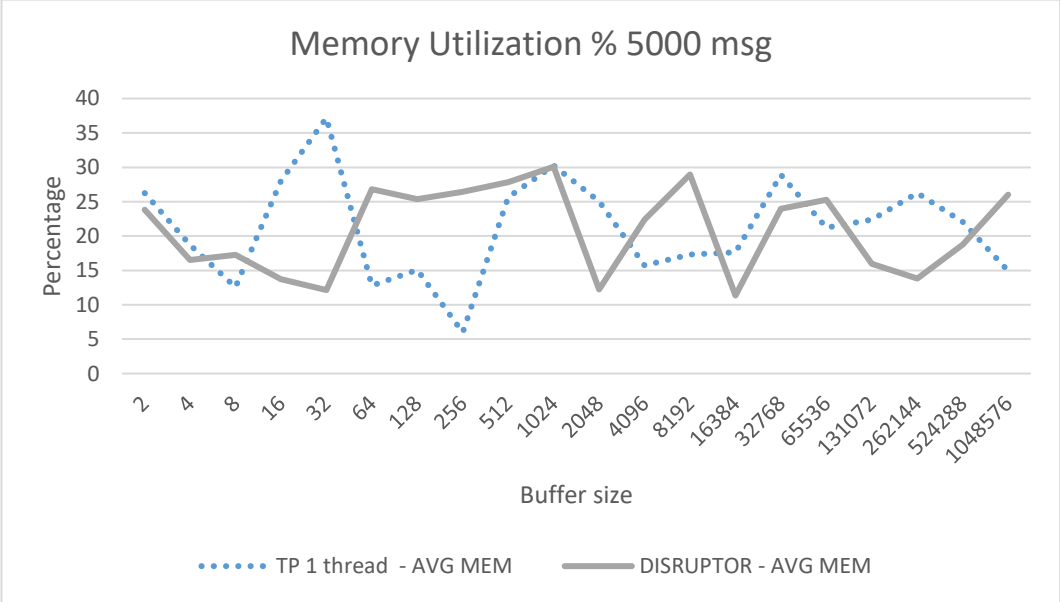
Gambar 4.13 CPU *utilization* untuk 120000 *message*s

Dalam penggunaan CPU, untuk jumlah *message* 1000, Threadpool menunjukkan awalan yang mirip dengan Disruptor, dengan selisih hanya sekitar 5% dan menemukan titik temu pada *buffer size* 1024. Seiring dengan penambahan jumlah *message*, Threadpool mengalami awalan yang lebih tinggi daripada Disruptor dengan selisih skitar 30% dan seiring dengan peningkatan jumlah *buffer*, nilai penggunaan CPU Threadpool mengalami penurunan dan menemukan titik temu dengan Disruptor pada jumlah *buffer* tertentu. Jumlah *buffer* teserbut sama dengan jumlah *buffer* yang diperlukan untuk mencapai *success rate* 100%.

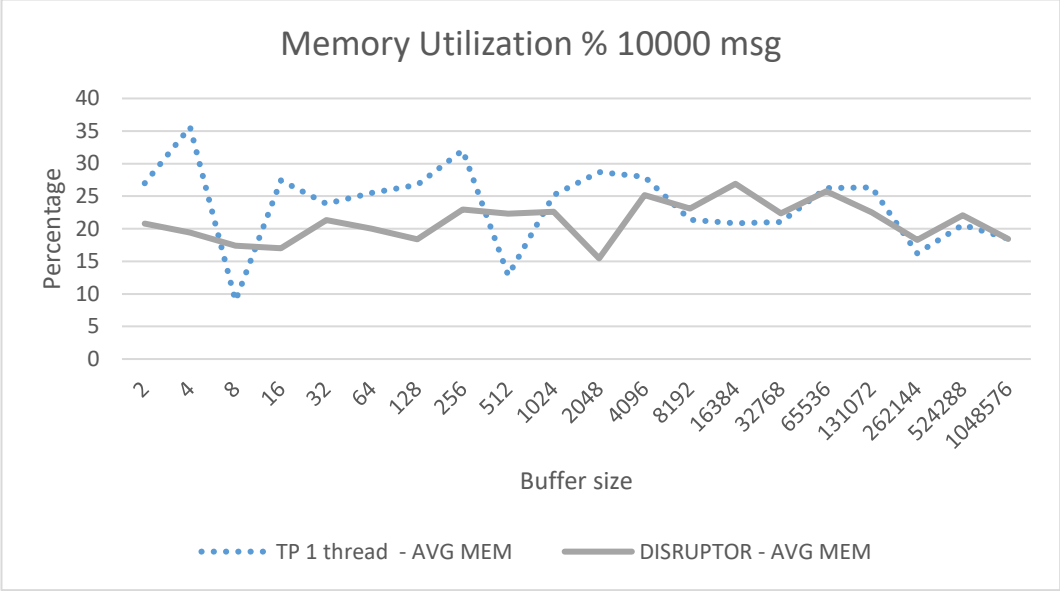
1.2.3 Memory *utilization*



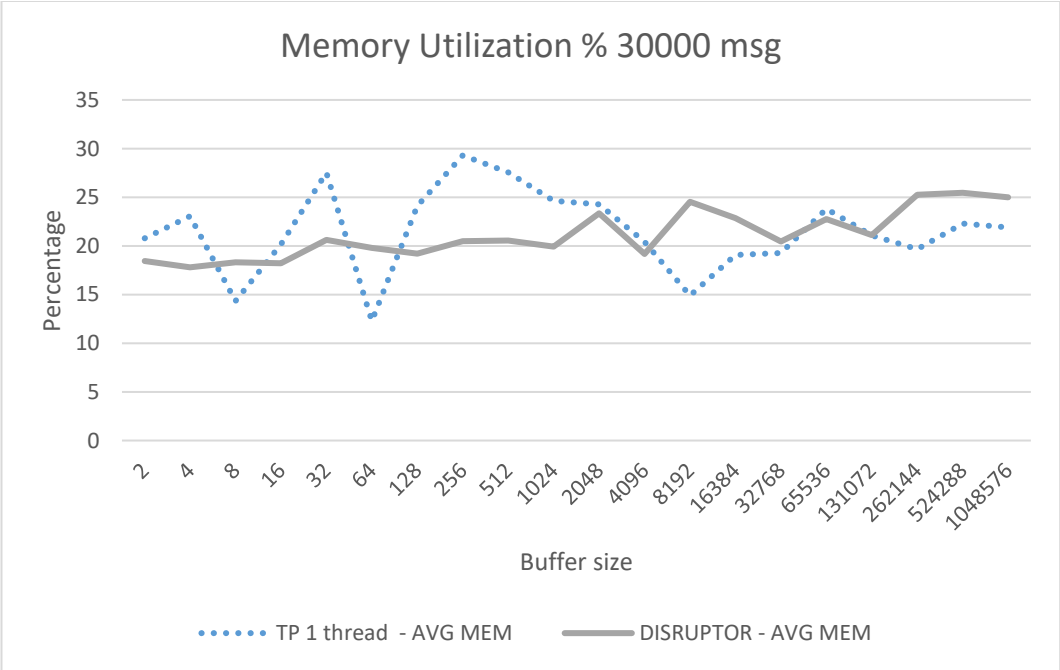
Gambar 4.14 Memory *utilization* untuk 1000 *message*s



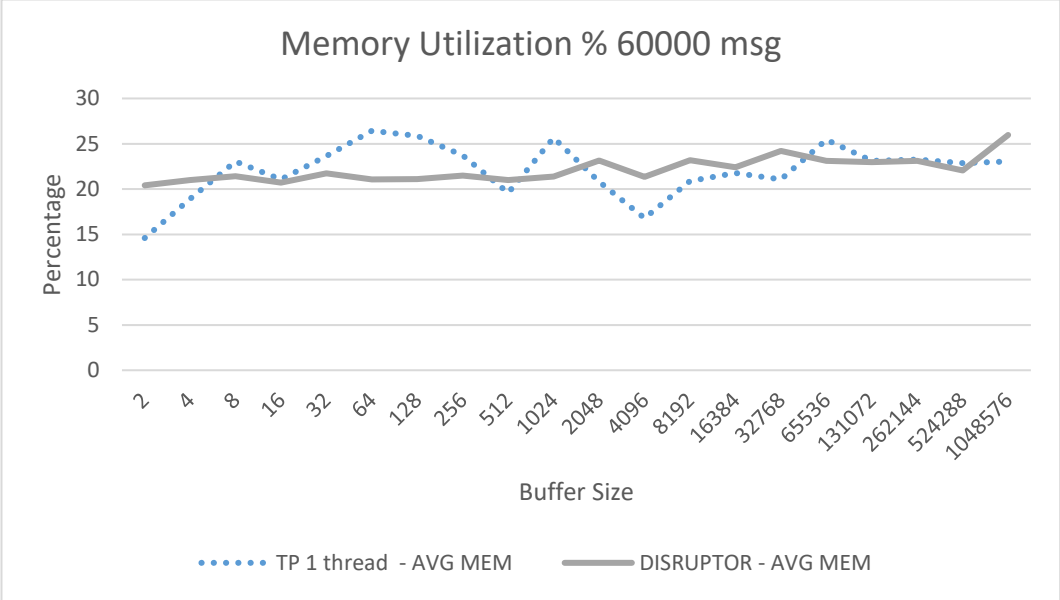
Gambar 4.15 Memory *utilization* untuk 5000 *message*s



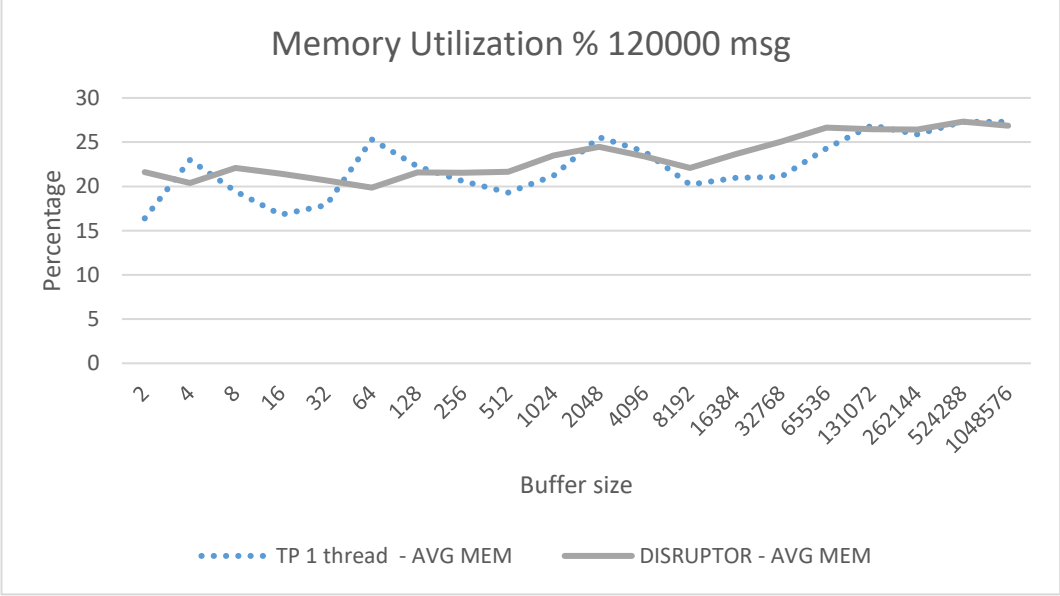
Gambar 4.16 Memory *utilization* untuk 10000 *message*s



Gambar 4.17 Memory *utilization* untuk 30000 *message*s



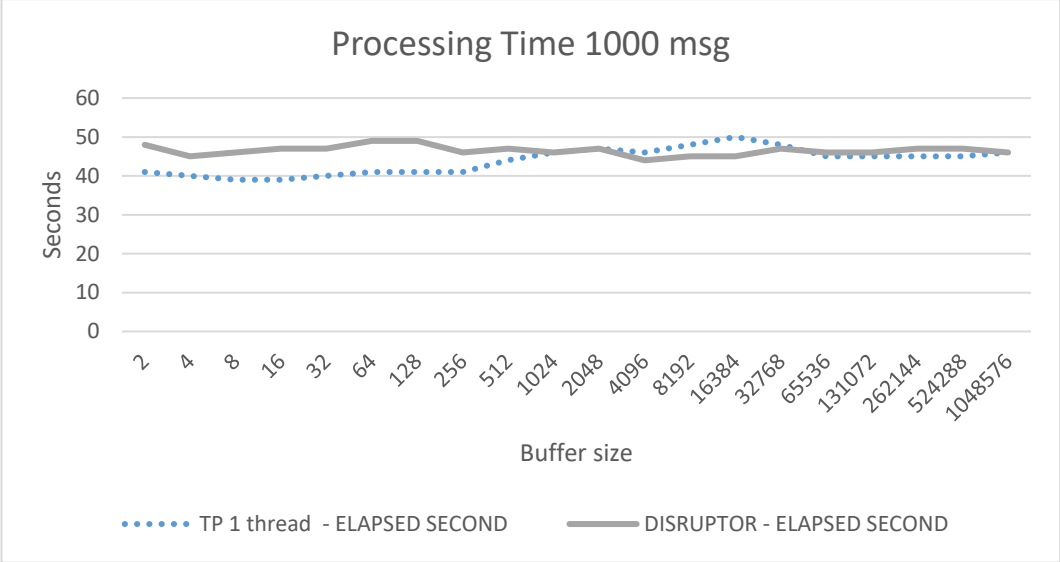
Gambar 4.18 Memory *utilization* untuk 60000 *message*s



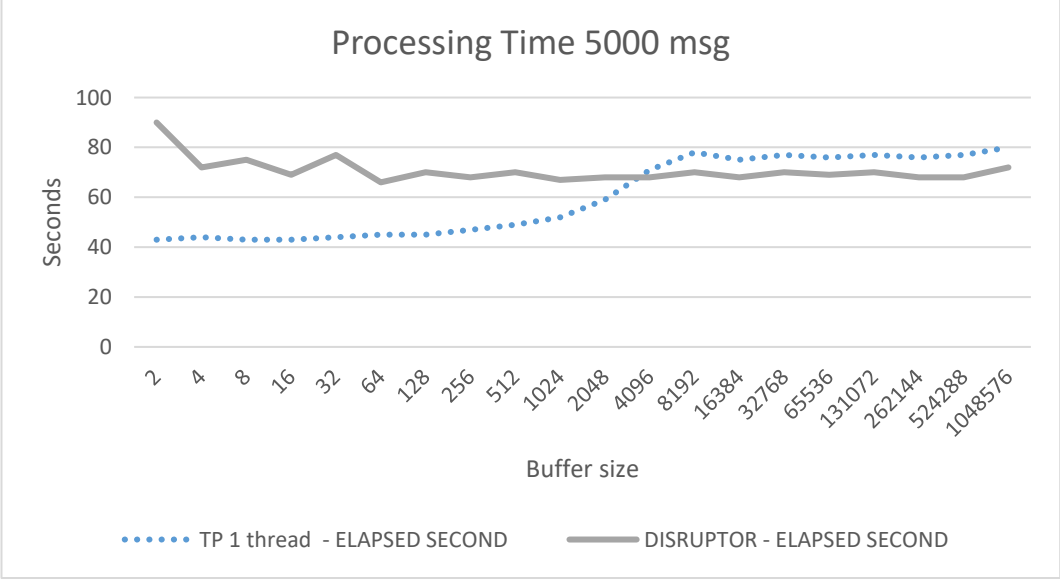
Gambar 4.19 Memory *utilization* untuk 120000 *message*s

Sedangkan untuk penggunaan *memory*, tidak ditemukan adanya penggunaan yang signifikan baik terhadap Threadpool maupun Disruptor

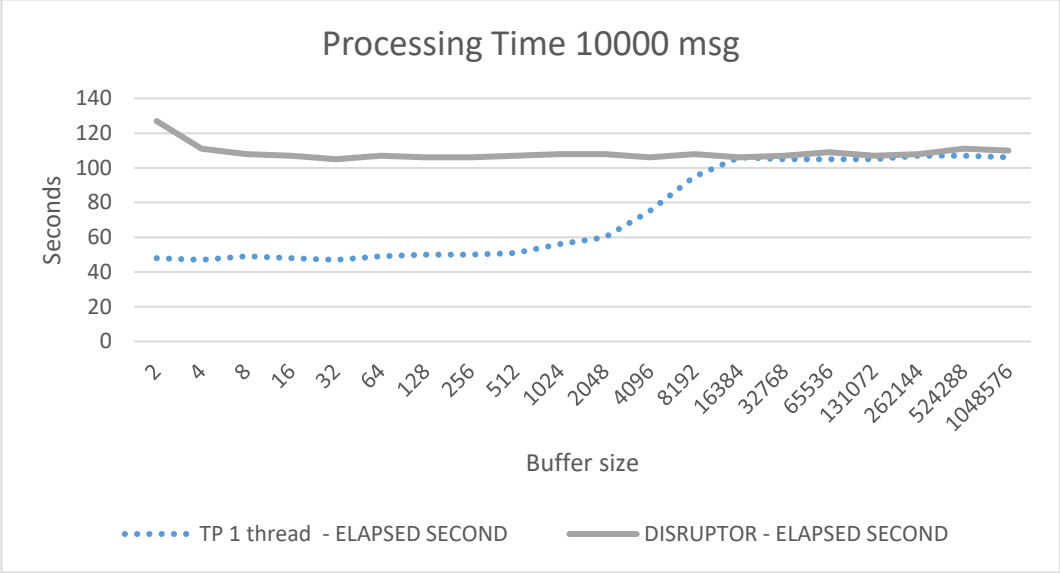
1.2.4 Elapsed times



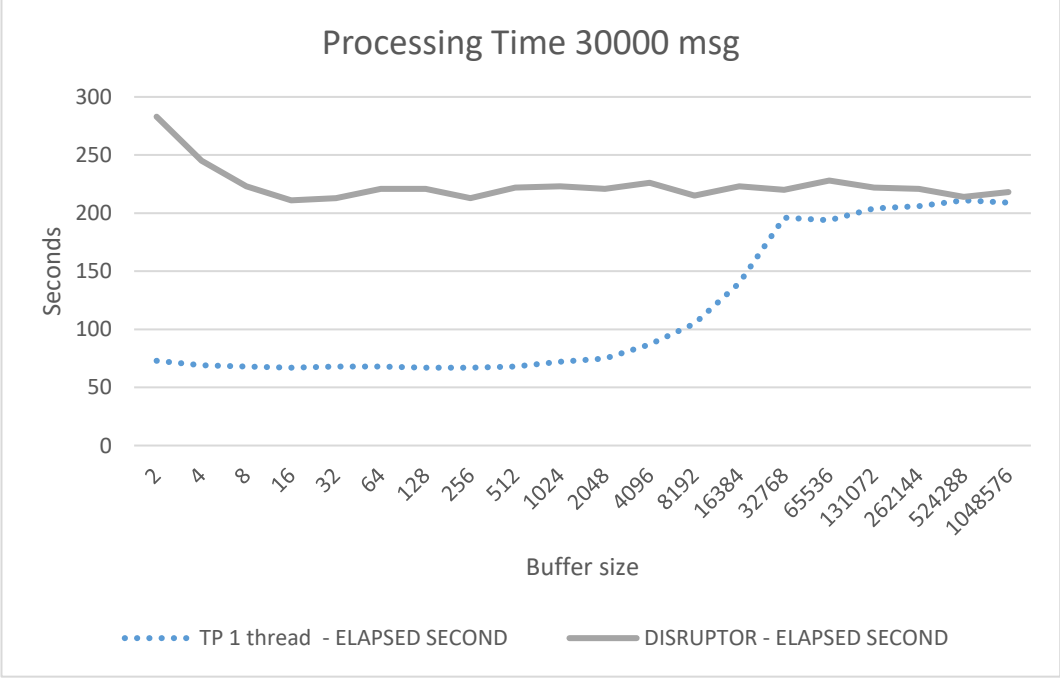
Gambar 4.20 Waktu untuk pemrosesan 1000 *message*s



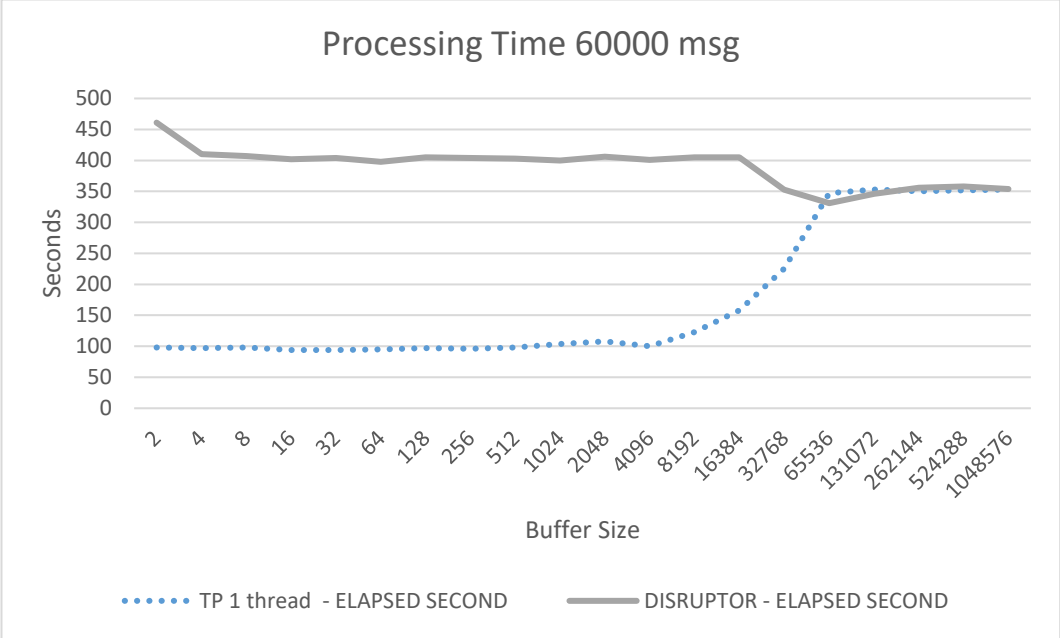
Gambar 4.21 Waktu untuk pemrosesan 5000 *message*s



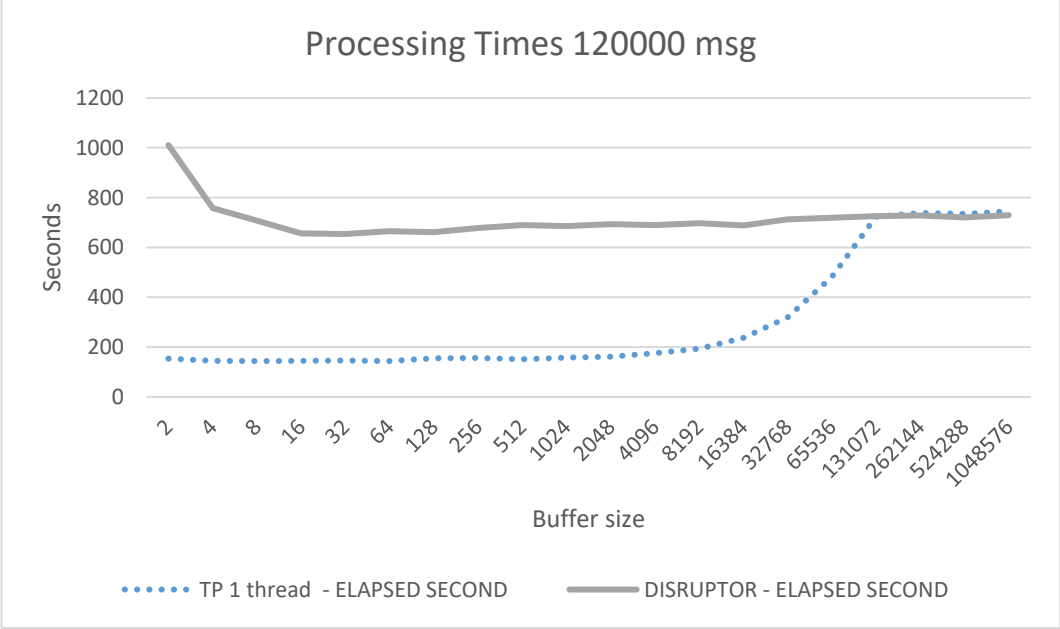
Gambar 4.22 Waktu untuk pemrosesan 10000 *message*s



Gambar 4.23 Waktu untuk pemrosesan 30000 *message*s



Gambar 4.24 Waktu untuk pemrosesan 60000 *message*s



Gambar 4.25 Waktu untuk pemrosesan 120000 *message*s

Pada jumlah *message* 1000, Threadpool dan Disruptor terlihat start dengan awalan yang mirip, dan dengan selisih yang tidak terlalu banyak, namun seiring pertumbuhan jumlah *message*, selisih menjadi lebih signifikan (30 detik, 60 detik

dan tekhir pada *message* 120000, menjadi 400 detik), namun, hal ini berbanding lurus dengan *success rate* yang didapat.

Dari proses – proses yang telah dilakukan dalam penelitian ini, terdapat beberapa hal yang dapat disimpulkan.

LMAX Disruptor memiliki *success rate* 100% dengan konsekwensi waktu pemrosesan yang lebih lama, karena ia menerapkan *blocking process* tidak seperti Threadpool yang langsung menolak *task* ketika *buffer size* telah penuh, sehingga LMAX Disruptor cocok digunakan pada sistem yang mengutamakan *success rate* tanpa memperhatikan waktu pemrosesannya, sebaliknya Threadpool memiliki waktu pemrosesan yang cenderung lebih cepat untuk pemrosesan *message* di bawah 10 ribu, namun untuk menanggulangi kehilangan *message*, dibutuhkan usaha tambahan dalam tahap implementasi.

Kedua *framework* ini tidak memengaruhi *memory* secara signifikan, karena baik Threadpool maupun LMAX Disruptor menitikberatkan pemrosesan dengan *thread*.

Threadpool lebih mudah digunakan dalam *development* karena *scope*nya yang lebih kecil daripada LMAX Disruptor namun untuk pemrosesan berskala besar yang membutuhkan konfigurasi dan integrasi yang bersifat *scalable*, LMAX Disruptor lebih unggul, namun dibutuhkan desain sistem yang lebih rumit.

LMAX Disruptor cocok digunakan untuk pemrosesan *message* berbasis prioritas karena sifatnya yang dapat mengakses *ring buffer* secara random berdasar index, tidak seperti Threadpool yang menggunakan *queue* (FIFO).