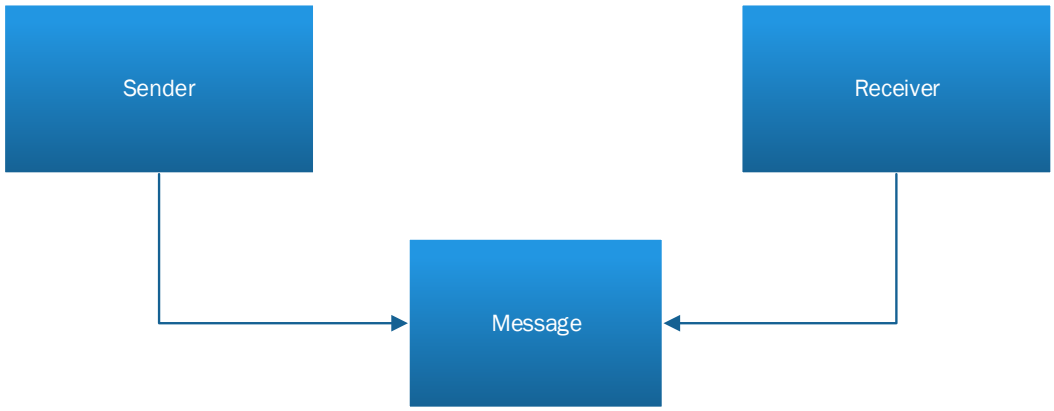
BAB II

TINJAUAN PUSTAKA

1.1 Tinjauan Pustaka

1.1.1 Messaging

Messaging adalah sebuah konsep dalam pengembangan aplikasi terintegrasi, masalah terutama di dalam aplikasi berskala besar adalah bagaimana cara berkomunikasi satu sama lain, terutama apabila aplikasi – aplikasi tersebut tidak memiliki format maupun standard pengiriman input yang sama. Seperti yang disebutkan pada [4], messaging adalah sebuah teknologi yang memberikan komunikasi antar aplikasi secara cepat dan *asynchronous*.



Gambar 2.1 Contoh dari komunikasi menggunakan *message*

Tidak ada aturan resmi bentuk dari sebuah *message*, karena sangat bergantung pada desain dan kebutuhan dari aplikasi yang ada, namun pada

dasarnya, sebuah *message* didefinisikan sebagai sebuah struktur data yang umumnya terdiri dari:

1. *Header*, berisi informasi tentang *message* itu sendiri, seperti siapa pengirim, tujuan dan deksripsi lain yang dibutuhkan aplikasi yang menggunakan *message* ini.
2. *Body*, berisi data sebenarnya yang hendak dikirim, bagian ini dapat berupa banyak hal, seperti instruksi untuk melakukan pekerjaan selanjutnya, maupun hanya sekedar notifikasi saja. Pada implementasi teknis, wujud yang umum adalah satu atau lebih object dalam bentuk *collection* (contohnya adalah *Array* maupun *Map*).

Contoh dari *message* sederhana adalah sebuah *string* yang merepresentasikan nilai tertentu bagi aplikasi penerima, seperti berikut:

"This is a sample message from your ISP, please ignore this, thank you"

Contoh yang lebih kompleks dapat dilihat pada *SOAP Message* berikut, yang berisi *header* dan *body*.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
      <t:Transaction
         xmlns:t="some-URI"
         SOAP-ENV:mustUnderstand="1">
            5
      </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DEF</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Gambar 2.2 SOAP *message* seperti yang tercantum pada [4]

Dengan menggunakan konsep ini, beberapa keuntungan yang diperoleh adalah:

1. Memungkinkan terjadinya komunikasi jarak jauh antar aplikasi, persoalan yang umum ditemukan di lapangan adalah aplikasi – aplikasi yang berkomunikasi tidak selalu terletak di dalam satu lingkungan jaringan yang sama.
2. Aplikasi memiliki format, struktur data dan protokol pengiriman yang berbeda – beda, sehingga dibutuhkan suatu bentuk yang "dikenali" oleh aplikasi – aplikasi yang saling terintegrasi, sistem yang menggunakan *message* dapat menjadi perantara dan berperan sebagai penerjemah.

3. Sebuah *message* dapat dianggap sebuah *task* yang berisi instruksi untuk penerimanya, apabila sebuah sistem melakukan instruksi tersebut dan tidak akan menerima instruksi baru sebelum yang lama selesai, tentunya hal ini akan menambah *overhead* dari sisi waktu pemrosesan dan kompleksitas desain. Dengan sistem *messaging* yang menekankan konsep *asynchronous*, pengirim tidak perlu menunggu selesainya *task* yang harus dilakukan, pengirim hanya perlu memastikan bahwa *message* sudah terkirim.
4. *Throttling*, konsep *messaging* juga memungkinkan untuk melakukan penyesuaian tergantung pada keadaan, contohnya adalah terlalu banyak *message* yang harus ditangani sebuah *receiver*, hal ini bisa diatasi dengan menambah maupun *tune up receiver* yang ada. Pada saat penyesuaian terjadi, *sender* tidak perlu mengetahui apapun yang terjadi sehingga ia dapat terus menjalankan tugasnya terlepas dari keadaan *receiver* (partial recovery).
5. *Fault recovery*, apabila terjadi *error* maupun kejadian yang membuat *sender, receiver* maupun keduanya tidak bekerja, *message* yang berada di "tengah perjalanan" tidak akan hilang karena adanya konsep *persistence storage* yang dimiliki oleh JMS Manager. Pada saat *sender* dan *receiver* berfungsi kembali, *message* akan kembali diproses seperti semula.

Selain sebagai *identifier*, *header* juga digunakan dalam kaitannya dengan keamanan data, namun hal tersebut tidak akan dibahas disini.

1.1.2 *Middleware* Berorientasi *Message*

Sistem *middleware* adalah sebuah lapisan tengah antara sistem operasi jaringan dan aplikasi terdistribusi [4] dan berfungsi sebagai perantara antara kebutuhan integrasi pada dua atau lebih sistem lain serta digunakan untuk pemrosesan data dalam jumlah dan variasi yang beragam. Salah satu implementasi dari sistem *middleware* adalah *message broker* yang dikenal juga dengan *middleware* berorientasi *message*, mendukung komunikasi antar sistem dengan mengirimkan *message* kepada sistem lain, yang dapat berisi perintah untuk permintaan fungsi tertentu atau dapat pula berupa data yang kemudian akan diolah berdasarkan aturan tertentu.

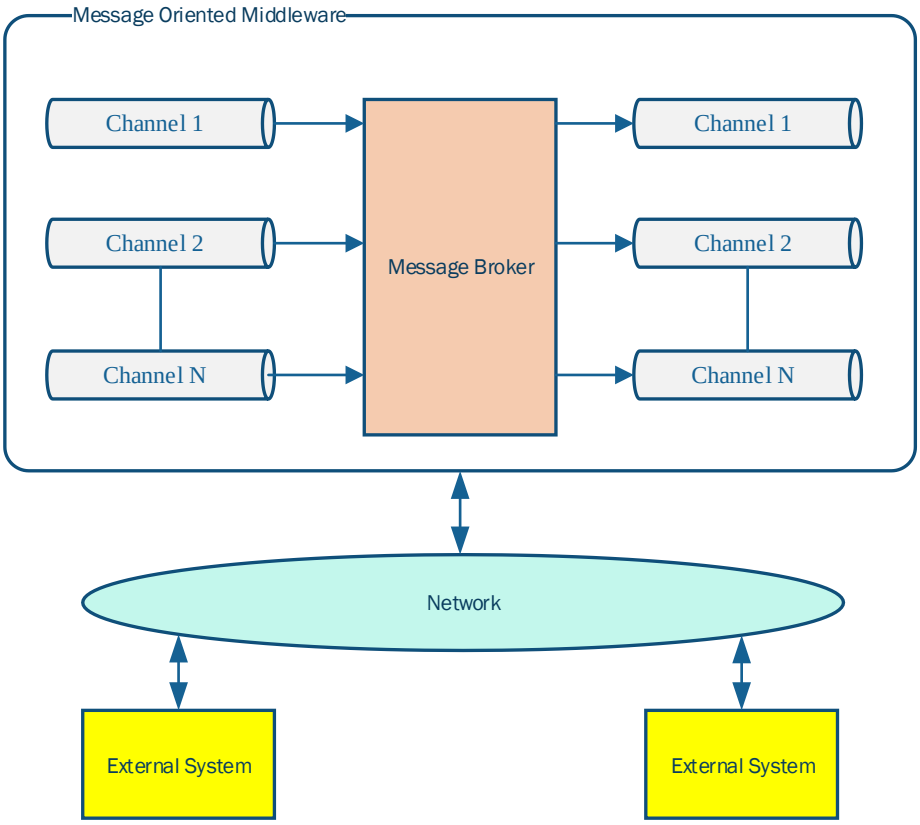
Menurut artikel [8], terdapat beberapa terminologi pada konsep sistem yang berbasis *messaging*, beberapa diantaranya yaitu:

1) *Channels*: Jalur virtual yang menghubungkan *sender* dan *receiver*, implementasi dari *channels* bervariasi dan tergantung dari kebutuhan serta arsitektur teknis aplikasi.
2) *Messages*: Paket data yang ditransmisikan melalui *channel*, apabila sebuah data akan dikirimkan, maka data tersebut akan "dibungkus" menjadi satu atau lebih *message* dan dikirimkan via *channel*, untuk kemudian *receiver* akan mengekstrak data dari *message* yang diterima. Proses ini biasanya disertai dengan mekanisme balasan (*response-reply*) *receiver* akan mengirimkan sinyal apabila *message*

sukses dikirimkan dalam jangka waktu tertentu atau pada jumlah percobaan pengiriman tertentu.

3) *Routing*: Implementasi pada kasus *large enterprise*, pada situasi nyata dapat terdapat banyak *channels* dan aplikasi, maka besar kemungkinan *message* harus melalui beberapa *channels* sebelum tiba pada tujuannya. Dengan adanya kompleksitas rute yang dapat terjadi, akan lebih baik apabila terdapat sebuah *message router*, komponen yang menentukan rute pada topologi *channel* yang ada dan mengarahkan *message* tersebut pada aplikasi tujuan ataupun kepada *router* lainnya berdasarkan aturan - aturan bisnis tertentu (*pre-defined rules*).
4) *Transformations*: Aplikasi – aplikasi yang terhubung pada sistem, memiliki beragam format dan dikirim melalui protokol yang berbeda pula. Untuk mengatasi masalah ini, *message* yang diterima oleh aplikasi MoM harus melalui *intermediate filter* yang mengubah format *message* tersebut menjadi sebuah format standar di dalam ekosistem aplikasi itu sendiri, untuk kemudian dapat dimodifikasi lebih lanjut sesuai dengan kebutuhan bisnis yang ada.

Secara garis besar, MoM dapat dijelaskan seperti pada gambar berikut,



Gambar 2.3 *Middleware* berorientasi *message*

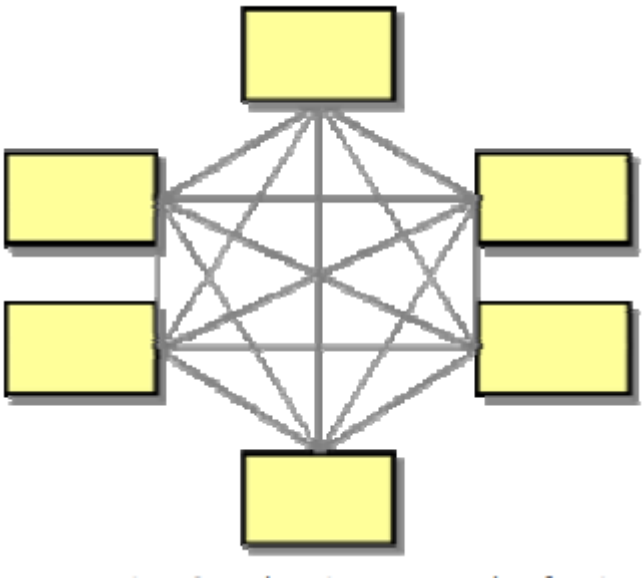
Seperti yang telah disinggung sebelumnya, skenario kasus dunia nyata yang rumit, melibatkan lebih dari satu *channel* sehingga diperlukan aturan bisnis tentang rute dari *message* yang biasanya ditentukan dari awal dan bersifat mudah dikonfigurasi. Contoh dari aturan bisnis ini adalah, apabila *message* diterima dari aplikasi A dan memiliki isi berawalan "*selamat pagi*" maka ubah isinya menjadi "selamat malam dunia" lalu kirim ke aplikasi B sekaligus kirimkan metadatanya ke aplikasi C, namun apabila *message* tersebut memiliki *identifier* yang berisi *string* "SX" maka drop *message* tersebut.

Beberapa keuntungan yang ditonjolkan dengan menggunakan arsitektur ini, seperti disebutkan oleh [15] adalah dengan desain komponen internal yang berdiri sendiri dan memiliki peran yang spesifik, maka ketika salah satu aplikasi yang terhubung mengalami malfungsi, tidak terlalu berdampak pada operasi aplikasi lain yang terhubung pada MoM. Hal ini mungkin dicapai dengan desain terpusat yang menerapkan setiap *channel* memiliki sebuah komponen sebagai *handler* (gambar 2.3). Selain itu, dengan adanya mekanisme *store and forward*, memungkinkan untuk *message* dipastikan terantar pada tujuannya, bahkan ketika terjadi kerusakan sebagian (*partial error*) karena *message* yang dikirimkan akan terlebih dahulu disimpan pada *persistence storage*, sehingga apabila aplikasi tujuan dalam keadaan tidak aktif, *message* tersebut tidak akan hilang di perjalanan dan akan langsung dikirimkan dari *persistence storage* apabila aplikasi klien kembali aktif.

Berhubung arsitektur ini mempromosikan *loose coupling*, yaitu setiap komponen dalam satu sistem memiliki perannya masing – masing secara spesifik dan tidak ataupun memiliki sedikit sekali pengetahuan mengenai komponen lain yang terpisah, maka menjadi relatif lebih mudah untuk melakukan modifikasi maupun perbaikan apabila terjadi masalah maupun modifikasi fitur. Sebagai contoh, pada MoM terdapat komponen *channel listener* yang hanya bertanggung jawab terhadap *message* yang datang pada *channel* tertentu dan terdapat *transformator* yang hanya bertugas memodifikasi *message* berdasarkan aturan tertentu.

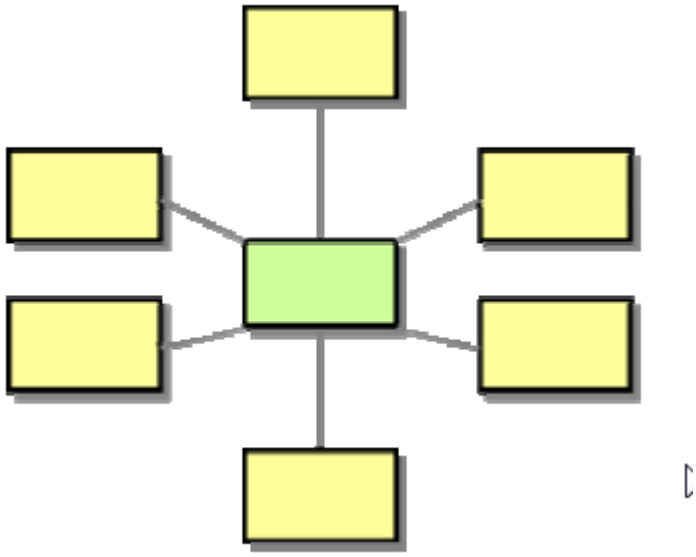
Desain ini menimbulkan permasalahan yang dapat terjadi pada tahap implementasi, yaitu *point to point connections* antar komponen seperti yang terlihat

pada gambar di bawah ini, yang mana hal tersebut dapat menambah kerumitan implementasi dan perawatan aplikasi MoM itu sendiri.



Gambar 2.4 Integrasi Model *Spaghetti* pada *Point to Point Connections* [4]

Untuk mengatasi permasalahan di atas, dibutuhkan sebuah komponen yang bertanggung jawab menentukan rute dari semua *message* yang masuk ke dalam aplikasi MoM tersebut, sehingga dibuatlah desain MoM yang bersifat terpusat. Dengan desain ini, maka diharapkan dapat meminimalisasi usaha untuk modifikasi fitur pada aplikasi MoM, seperti penambahan dan pengurangan channel listener, komponen internal baru, dan sebagainya. Desain ini menjadikan router sebagai titik pusat dari kegiatan utama yang dilakukan oleh MoM, yaitu mengarahkan *message* pada aplikasi tujuan sesuai dengan rutenya, seperti yang ditunjukan pada gambar di bawah ini.

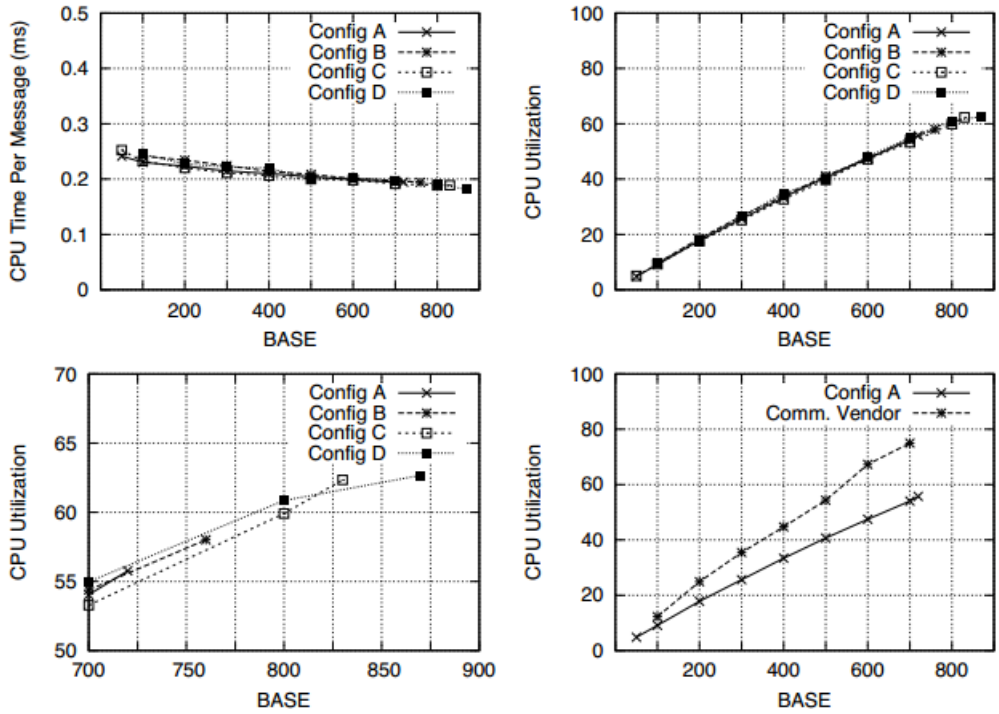


Gambar 2.5 *Message* Broker Terpusat [4]

1.1.3 Pengukuran Kinerja JMS

Penggunaan *middleware* berorientasi *message* (MoM) pada layanan – layanan inti industri finansial dan telekomunikasi seringkali menghadapai tantangan – tantangan serius dalam adaptasinya terhadap perkembangan teknologi dan kebutuhan yang menuntut penyesuaian kinerja yang lebih baik. Artikel [12] menyebutkan bahwa terdapat beberapa parameter pengukuran kinerja JMS, artikel tersebut juga menyebutkan bahwa model *messaging* itu sendiri dapat dibagi menjadi *point to point* (P2P) dan *publish subscribe*. Spesifikasi JMS juga menetapkan beberapa jenis pengantaran *message* berdasarkan QoS (*Quality of Service*) yang berbeda, yaitu *non-persistent / persistent, non-durable / durable, non-transactional / transactional*. Adapun pengukuran kinerja JMS melibatkan

beberapa parameter antara lain adalah ukuran *message*, *message thoughput* (*message*/satuan waktu), CPU maupun memory *utilization* (%).



Gambar 2.6 Contoh pengukuran kinerja ActiveMQ dengan SPECjms2007 [12]

Sebagai contoh, pada gambar di atas, dilakukan pengujian untuk mengukur CPU *utilization* terhadap ukuran *message* (kilobyte) berdasarkan konfigurasi tertentu (A, B, C dan D). Dari sini kita dapat mengambil kesimpulan konfigurasi mana yang paling baik berdasarkan CPU *utilization* yang paling rendah dan juga melihat bahwa semakin besar ukuran suatu *message*, maka akan berbanding lurus dengan konsumsi CPU yang digunakan, tentunya dalam *scope* konfigurasi yang diberikan.

1.1.4 COST

Eksperimen yang dilakukan pada artikel [10], menyatakan bahwa pada sistem dengan pemrosesan paralel, belum tentu memiliki kinerja yang lebih baik dibandingkan dengan menggunakan *single thread*. Terkait dengan hal ini, maka diperkenalkanlah istilah COST atau *Configuration that Outperform Single Thread* sebagai metrik baru dalam platform *big data* yang membandingkan skalabiliitas sebuah sistem terhadap performance yang dihasilkan. Percobaan yang dilakukan pada artikel ini menjadi salah satu faktor pendukung dilakukannya penelitian ini.

Penelitian yang dilakukan adalah melakukan percobaan ulang dari beberapa *graph processing paper* yang melaporkan percobaan menggunakan framework dan algoritma yang beragam, dengan sebuah algoritma sederhana berbasis *single thread* dengan set yang sama.

name	twitter_rv [11]	uk-2007-05 [4]
nodes	41,652,230	105,896,555
edges	1,468,365,182	3,738,733,648
size	5.76GB	14.72GB

Gambar 2.7 Dataset yang digunakan dalam eksperimen COST **[10]**

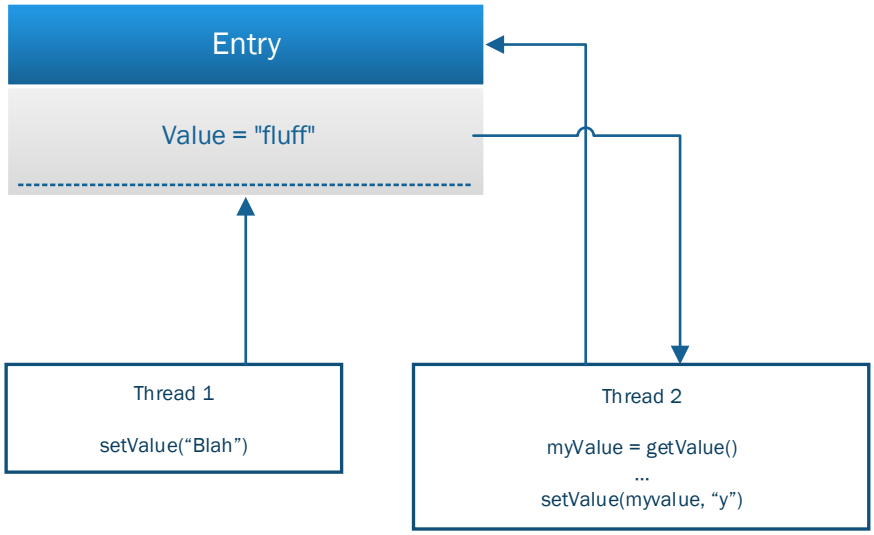
scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Gambar 2.8 Hasil perbandingan pengukuran dalam eksperimen COST [10]

Didapat hasil bahwa pemrosesan menggunakan *single thread* dapat mengungguli waktu pemrosesan oleh beberapa kerangka kerja pemrosesan paralel, hal ini menjadi tantangan tersendiri dan memunculkan pertanyaan baru. Adapun alasan yang mungkin menjadi penyebab anomali ini adalah sebuah *framework* mungkin baik pada suatu kondisi tertentu saja sehingga tidak dapat dijadikan tolak ukur absolut dari solusi masalah yang ada. Selain itu mungkin terdapat faktor kualitatif lain seperti kemudahan dalam implementasi dan integrasi.

1.1.5 Concurrent Processing

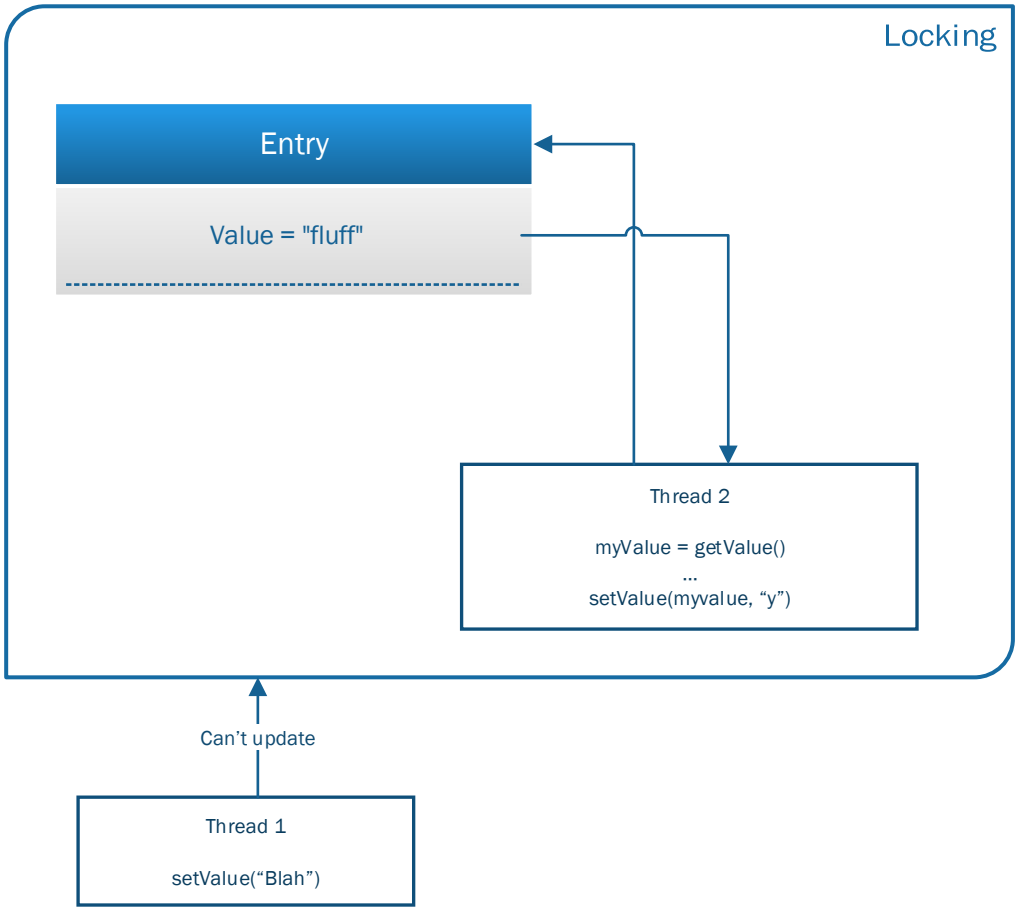
Pemrosesan paralel memiliki permasalahan tersendiri [19], terutama pada akses *resources* (*database*, *file*, *socket*, dll), inti dari pemrosesan itu sendiri adalah *mutual exclusion* dan *visibility of change*. *Mutual exclusion* berfokus pada pengaturan saat satu *thread* meng-update *shared resource* sehingga tidak terjadi *inconsistency* data, sehingga *thread* lain harus dapat melihat proses *update* yang sedang dilakukan, hal ini disebut *visibility of change*.



Gambar 2.9 Akses pada *shared resources*

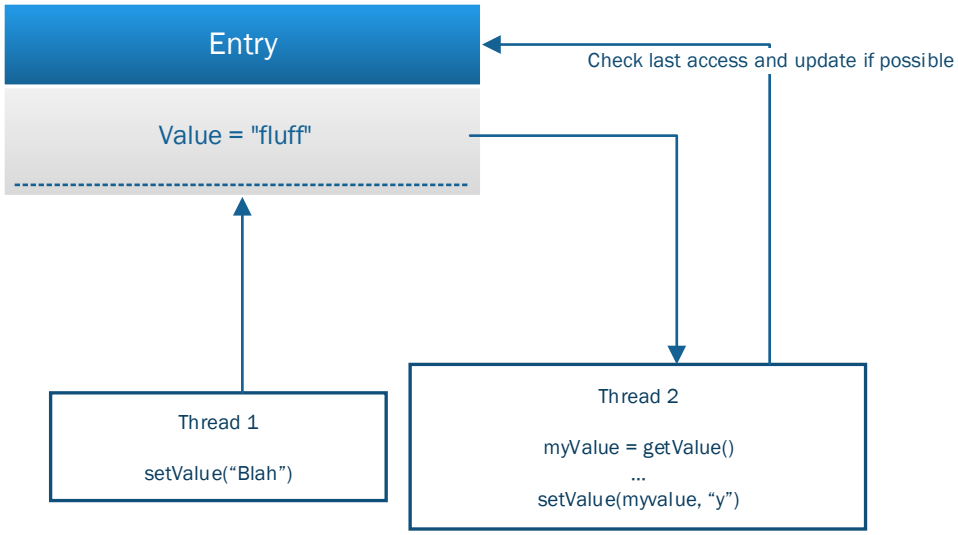
Instruksi paling *costly* pada lingkungan paralel adalah *write access*, terlebih apabila terdapat *multi threads* melakukan *write* pada *resource* yang sama, hal ini memerlukan koordinasi yang rumit dan *costly*. Salah satu cara untuk mencapai hal ini adalah dengan menggunakan prinsip *locking*.

1.1.5.1 Mekanisme *Lock*



Gambar 2.10 *Pessimistic Locking*

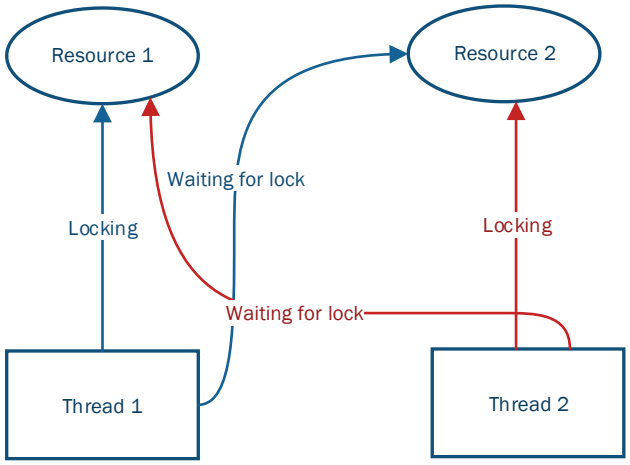
Seperti yang dapat dilihat pada gambar 2.10, pendekatan pertama adalah *pessimistic locking*, *thread* 2 lebih dahulu mengklaim entry dan ketika terjadi proses update, *thread* 2 melakukan *locking* terhadap *resource* tersebut sehingga ketika *thread* 1 hendak melakukan akses, proses tidak dapat berlanjut karena *lock* dipegang oleh *thread* 2. Kekurangan dari cara ini adalah potensi *hanging thread* yang terjadi apabila *thread* 2 terus mengklaim *entry* sehingga *thread* lain tidak dapat giliran mengakses, hal ini memiliki kemungkinan lebih dapat terjadi dengan semakin banyaknya jumlah *thread*.



Gambar 2.11 *Optimistic Locking*

Alternatif lainnya adalah dengan *optimistic locking*, *thread* 2 akan mengklaim *lock* pada *entry* hanya ketika akan melakukan *write*, *thread* 2 akan melakukan pemeriksaan dahulu apakah nilai dari *entry* sudah berubah sejak terakhir akses. Apabila ada *thread* lain yang melakukan *update* pada *entry*, otomatis *thread* 2 tidak dapat melakukan *update* karena nilainya berbeda dengan yang terakhir *thread* 2 kenali. Sehingga *thread* 2 perlu melakukan *retry* atau memberikan semacam *error message*. Salah satu implementasi prinsip ini adalah pada CAS (*Compare and Swap*).

Potensi masalah yang umum adalah *deadlock*, yaitu *thread* tidak dapat mengakses *resource* yang dibutuhkan karena *lock* dipegang oleh *thread* lain. Ilustrasi masalah ini dapat dilihat pada gambar di bawah



Gambar 2.12 *Deadlock*

Masalah yang lain adalah *lock* memiliki *overhead* waktu, karena *lock* membutuhkan keterlibatan OS untuk mediasi argument, dalam proses ini terjadi *context switching*. Dalam sistem yang membutuhkan *latency* kecil (*real time system*), desain untuk hal ini sangat diperlukan karena dapat memiliki dampak kontinu pada bagian sistem lain dan pada akhirnya dapat memperlambat sistem, apalagi saat implementasi pada *dirty code*.

Tim dari LMAX melakukan eksperimen dengan program sederhana yang meng-*increment* 64-bit *counter* dalam pengulangan sebanyak 500 juta kali. Eksperimen ini menunjukkan bahwa mekanisme *locking* memiliki waktu proses yang lebih lama dibandingkan dengan proses menggunakan *single thread* dan CAS serta kombinasinya, seperti yang ditunjukkan pada gambar di bawah

Method	Time (ms)
Single thread	300
Single thread with lock	10,000
Two threads with lock	224,000
Single thread with CAS	5,700
Two threads with CAS	30,000
Single thread with volatile write	4,700

Gambar 2.13 Perbandingan waktu proses *increment counter* 64 bit [19]

1.1.5.2 Queue

Permasalahan pada *queue* adalah pada saat menyimpan dan mengambil data di *queue*, kita harus melakukan *write*, hal ini menciptakan kondisi yang disebut *write contention*. Lebih dari satu *client* hendak mengakses satu *resource* yang sama, untuk mengatasi hal ini, *queue* menggunakan mekanisme *locking*, tetapi dengan digunakannya *locking* maka terjadi *context switching* pada *kernel*. Salah satu faktor dominan dengan *latency* adalah bagaimana CPU berinteraksi dengan *memory*, akses CPU ke *cache* lebih cepat daripada akses ke *memory*. Sehingga upaya peningkatan kinerja dapat dilakukan dengan berusaha supaya data yang sering diakses disimpan dalam *cache*.

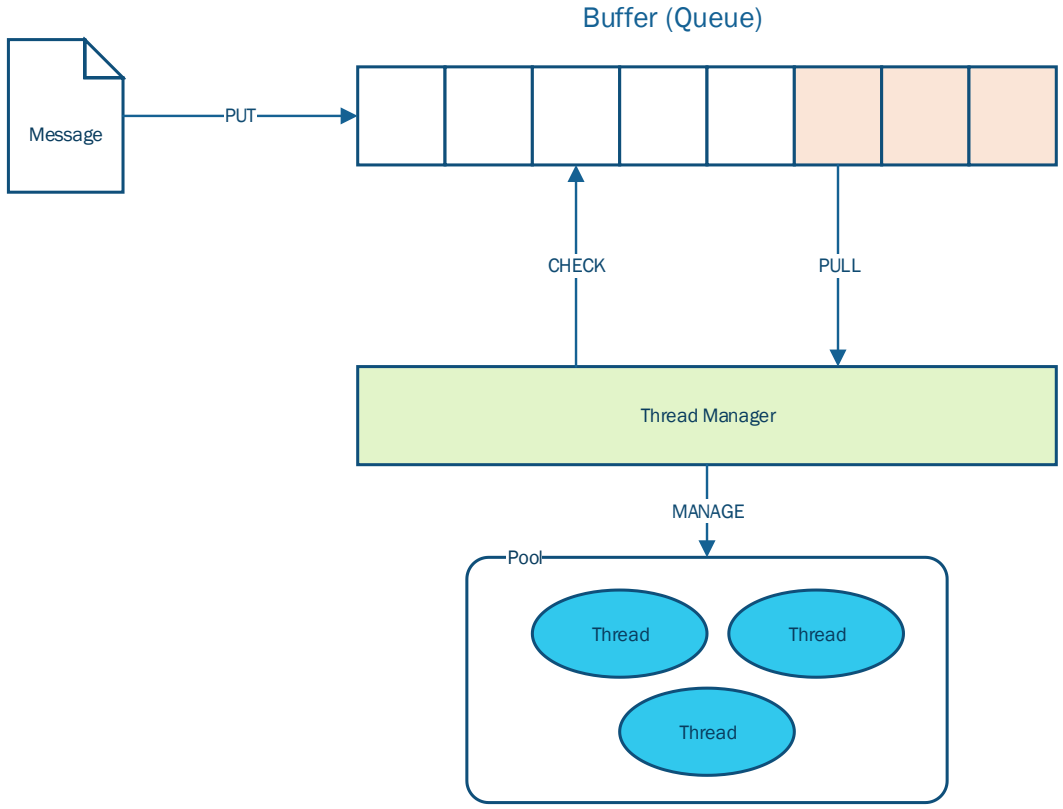
Queue cenderung memiliki *write contention* pada variabel *head*, *tail* dan *size* dikarenakan digunakannya *single large grain of lock* pada sebuah *queue* secara keseluruhan dan berpengaruh pada thoughput yang dihasilkan. Implementasi ini memiliki masalah selain kompleksitasnya, yaitu pada Java, *queue* adalah objek signifikan untuk *garbage collection* dimana hal ini dapat berkontribusi memberikan *latency*.

1.1.6 Threadpool

Implementasi dari sistem berskala besar dengan proses komputasi paralel yang tinggi, membutuhkan banyak penyesuaian teknis demi mencapai skalabilitas dan kinerja tertentu. Salah satu diantaranya adalah penggunaan Threadpool, menurut artikel [17], Threadpool adalah abstraksi dari wadah yang membuat dan mengatur siklus hidup dari *thread* – *thread* di dalamnya.

Pada saat sistem hidup, akan dibuatlah sebuah *pool* yang merupakan *container* untuk menampung sejumlah *thread* yang *stand by*. Pada saat *task* datang, *task* akan dimasukkan ke dalam sebuah *bounded buffer* (*working queue* / *task queue*). Apabila masih terdapat *slot* kosong di dalam *queue*, maka *task* akan langsung disimpan, namun apabila *slot* minimum sudah terisi (nilai awal adalah 1), maka *queue* akan memanjang sebanyak 1 *slot* dan menyimpan *task*. Apabila *queue* sudah mencapain batas *slot* maksimum, maka *task* yang datang akan langsung ditolak, hal ini dapat dilihat dalam bahasa pemrograman Java pada bagian *TaskRejectedExecutionException*. *Task* yang ada di dalam *queue* akan di-*consume* oleh *thread* yang *standby*, apabila jumlah *thread* minimum tidak dapat memenuhi laju pemrosesan *task*, maka *pool* akan membuat (*spawning*) *thread* baru untuk memroses. Hal ini akan berlanjut hingga mencapai batas maksimum *thread* yang diijinkan di dalam *pool*. Ketika sebuah *thread* selesai melakukan tugasnya, ia akan masuk ke dalam tahap *idle* / *stand by* dan ketika tidak ada *task* yang harus dikerjakan, maka ia akan mati (*terminate*). *Pool* bertugas untuk mengelola siklus hidup *thread* yang ada di dalamnya, termasuk *thread* mana yang hidup, mati,

maupun *deadlock*. *Pool* akan menjaga supaya dalam keadaan *idle* sekalipun terdapat sejumlah minimum *thread* yang *stand by*.



Gambar 2.14 Desain Threadpool

Kelebihan yang ditawarkan oleh konsep ini di antaranya adalah sebagai berikut:

1. Pemrosesan paralel yang dilakukan tentunya dapat membuat waktu proses lebih cepat, sehingga semakin banyak *task* yang dapat diselesaikan dalam satu waktu.
2. Pengelolaan siklus hidup *thread* dan penggunaannya diatur oleh *pool* secara *blackbox* (walaupun terdapat beberapa konfigurasi

eksternal) sehingga mengurangi kompleksitas yang dihadapi dalam tahap implementasi, terutama dalam *error handling* maupun penanganan situasi khusus seperti *deadlock* dan mekanisme *locking*. Selain itu, *overhead* yang terjadi pada saat pembuatan *thread* dapat diminimalisir dengan menggunakan ulang *thread* yang ada.

3. Struktur yang relatif kecil untuk sebuah *library* sehingga bersifat *versatile* dan mudah diadopsi ke dalam kode maupun ke dalam kerangka teknis lain.

Adapun kekurangan dari konsep ini antara lain adalah:

1. Ketika *pool size* mencapai suatu batas tertentu, kinerja pemrosesan dari Threadpool akan mencapai titik puncak dan apabila *pool size* diperbesar, maka kinerja yang dihasilkan justru akan menurun, hal ini kerap terjadi pada aplikasi yang membutuhkan kebutuhan komputasi yang besar, sehingga OS akan memiliki waktu yang lebih sedikit untuk menjadwalkan *thread* yang aktif, sebaliknya apabila aplikasi tidak terlalu intensif dalam komputasi, maka aplikasi akan menghabiskan sebagian besar waktunya pada IO, sehingga OS mendapat kesempatan lebih untuk menjadwalkan *thread* yang aktif. Pada dasarnya sebuah *processor* hanya dapat menjalankan instruksi secara simultan sebanyak jumlah *core* nya.
2. Terkait pada poin di atas, dibutuhkan proses *tuning* secara berkelanjutan untuk menemukan konfigurasi yang sesuai dengan aplikasi dan spesifikasi *server* yang digunakan, karena *thread* dan

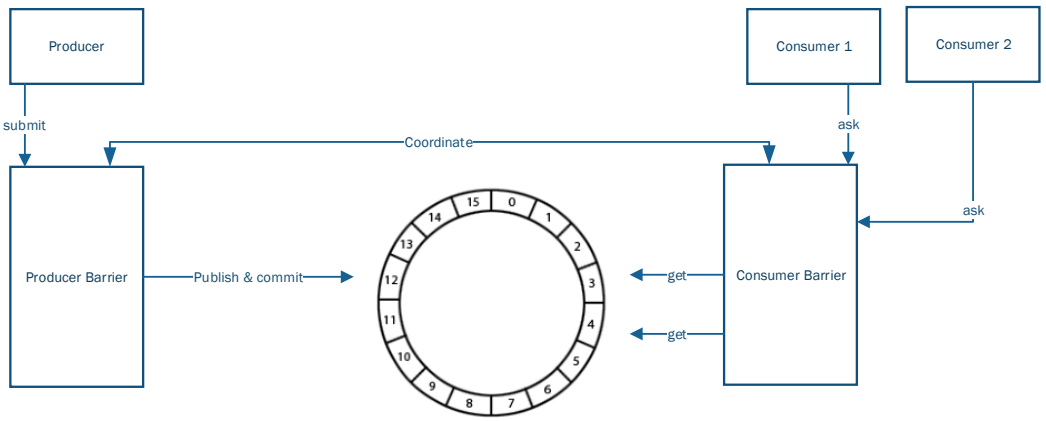
pool itu sendiri membutuhkan *resources* seperti *memory*, di luar dari pemrosesan yang dilakukan.

3. Dari sisi implementasi, terdapat kemungkinan penggunaan *resources* yang sama (*shared resources*) yang dapat menyebabkan *locking*, maupun *concurrent modification* sehingga menyebabkan *resources* tersebut menjadi inkonsisten. Selain itu, kode yang tidak rapi dapat berpotensi menyebabkan *infinite loop* sehingga *thread* tidak akan pernah selesai memroses *task*.

1.1.7 LMAX Disruptor

LMAX Disruptor adalah sebuah kerangka kerja teknis yang dibuat berdasarkan penelitian pada COST, dengan menekankan pada pemrosesan berbasis *single thread* per *consumer*, *framework* ini diklaim memiliki kinerja yang lebih baik dalam hal *latency* dan waktu pemrosesan dibandingkan *framework* sejenis seperti Threadpool.

Seperti yang ditunjukkan pada gambar 2.15, *framework* ini terdiri atas *producer barrier*, *consumer barrier* dan *ringbuffer* sebagai *container*. *Producer barrier* berfungsi mem-*publish* task ke dalam ring buffer dan *consumer barrier* berfungsi untuk mengambil task dari ringbuffer.



Gambar 2.15 *Producer Barrier* memastikan tidak ada *wrapping* seperti disebutkan dalam [12]

Producer barrier melakukan tugasnya yang terdiri dari tiga bagian yaitu mengklaim *next slot* pada *buffer*, mem-*publish* data dan akhirnya melakukan *commit*. Pada gambar 2.15 *producer barrier* mem-publish pada *slot* 1, namun ketika akan mem-*publish* pada *slot* berikutnya, *producer barrier* mengetahui bahwa *consumer* menempati *slot* itu dahulu, sehingga *producer* harus "menunggu" hingga *consumer commit* dan barulah ia dapat melakukan *claim slot* untuk mem-*publish* data. *Producer barrier* memiliki *consumer tracking* sehingga dapat mengetahui posisi *consumer*, ini sebagai cara agar *producer* tidak melakukan *over lapping* pada *ringbuffer*, karena Disruptor tidak memiliki *head* dan *tail pointer* seperti yang ada pada *queue*, dengan mekanisme ini, mengurangi *contention* yang terjadi pada *queue*. Ketika nilai baru telah di-*publish*, maka *producer barrier* melakukan *commit* dan *notify consumer barrier* bahwa ada nilai baru yang di-*publish*.

Consumer barrier memanggil *method* berikut untuk mengetahui *sequence* mana yang siap di-*consume*

```
final long availableSeq = consumerBarrier.waitFor(nextSequence);
```

Setelah mengetahui *sequence* yang *available*, *consumer barrier* akan mengambil nilai yang ada di dalam *ringbuffer* dan sekaligus *update cursor*.

Keunikan sekaligus kelebihan dari *framework* ini seperti yang disebutkan pada [19] antara lain adalah:

1. Alokasi *memory* untuk *ringbuffer* dilakukan saat sistem dimulai, dengan jumlah yang tetap dan berfungsi sebagai *container* untuk data yang disimpan. Dengan adanya pre-alokasi ini, dapat meminimalisir terjadinya *garbage collection*, karena *entry* akan terus "hidup" dan digunakan kembali berulang (*reuse*) selama *instance* Disruptor hidup.
2. Disruptor tidak menggunakan *locking*, melainkan CAS (*Compare and Swap*), seperti yang disebutkan pada pada [8], operasi CAS lebih hemat dibandingkan *locking* karena tidak melibatkan OS. Saat ada nilai yang di-*consume*, nilai tersebut di-*copy* dan bukan diambil, sehingga nilai baru akan meng-*overwrite* nilai lama. Penggunaan *ringbuffer* yang tidak memiliki "*end*" *pointer* seperti pada *queue* membantu mengurangi *contention* yang berkontribusi pada *latency jitter*.
3. *Self managed producer* dan *consumer barrier* secara otomatis melakukan *blackbox publish – consume* yang efektif.

4. Dengan penggunaan *single consumer*, CPU *utilization* diharapkan lebih rendah dibandingkan *framework* lain namun dengan waktu pemrosesan yang lebih cepat.

Adapun kekurangan dari *framework* ini antara lain adalah:

1. Implementasi dari *framework* ini membutuhkan desain yang lebih kompleks dibandingkan Threadpool, sehingga pada beberapa kasus implementasi ini akan dianggap "*overkill*".
2. Perlu adanya *tuning session* untuk menemukan konfigurasi yang tepat

1.1.8 Komparasi Threadpool dan LMAX Disruptor

Berikut adalah perbandingan dasar antara Threadpool dan LMAX Disruptor

Tabel 2.16 – Komparasi Threadpool dan LMAX Disruptor

Category	Threadpool	LMAX Disruptor
Single thread		V
Multi thread	V	
Pooling system	V	
No overhead in locking mechanism		V
No overhead in memory contention		V
Has buffer	V	V
FIFO access to buffer	V	V
Random access to buffer		V
Custom wait strategy	V	V

Berdasarkan tinjauan – tinjauan dari berbagai sumber di atas, LMAX Disruptor berusaha memperbaiki kekurangan yang terdapat pada Threadpool.

LMAX Disruptor tidak akan serta merta menggantikan posisi Threadpool, karena kompleksitas desain sistem akan jauh lebih rumit dengan menggunakan LMAX Disruptor sehingga hanya dapat mengisi posisi *parallel processing framework* pada beberapa bagian saja, seperti *routing* dan *mediation publishing*. Untuk pemrosesan paralel yang *scope*-nya lebih kecil dan dengan jumlah *task* tertentu masih dapat ditangani dengan baik oleh Threadpool.

Berdasarkan pertimbangan – pertimbangan di atas, maka akan sangat bermanfaat sekali apabila kita dapat mengkombinasikan kedua *framework* tersebut untuk mencapai pemrosesan paralel yang adaptif dalam menutupi kelemahan masing – masing *framework*. Kombinasi dilakukan dengan menggunakan Threadpool pada inisialisasi, karena *scope*-nya yang lebih kecil dan lebih sederhana, ketika sistem mencapai kondisi tertentu, maka sistem akan menggunakan LMAX Disruptor, hal ini dilakukan dengan dugaan bahwa untuk pemrosesan dalam jumlah *message* tertentu, Threadpool adalah pilihan yang baik untuk digunakan, namun apabila *message* sudah mencapai di atas jumlah yang dapat diproses Threadpool, maka LMAX Disruptor akan memiliki kinerja yang lebih baik.