Perancangan *System Crawler* dengan Menerapkan Arsitektur *Distributed Task*

http://dx.doi.org/10.28932/jutisi.v8i1.4205

Riwayat Artikel
Received: 26 November 2021 | *Final Revision*: 28 Maret 2022 | *Accepted*: 28 Maret 2022

Heri Santoso ✉ #1, Indyah Hartami Santi #2, Ni'ma Kholila #3

Teknik Informatika, Universitas Islam Balitar
Jalan Majapahit No 2-4, Sananwetan, Blitar, Indonesia
[1]herisantoso1998@gmail.com
[2]indyahhartami@gmail.com
[3]nimakholila@gmail.com

Abstract — **PDC Media Group is a company engaged in online trading. The need for data insight in the online marketplace is very important. Likewise, how to get quite a lot of data, of course, requires automation such as *crawling* data on the marketplace website. Due to the large amount of data, crawler systems are often not optimal in crawling data. The application of distributed tasks on the crawler system provides convenience in scaling the server both vertically and horizontally. Therefore, the large and growing data can be handled by the *crawler system*. The application was developed with the Python language, with the application server using Google Cloud Computing. In a distributed task architecture requires a component in the form of a message broker. The message broker used in designing this system is RabbitMQ. Testing the crawler system uses 3 scenarios, namely with 1 worker, 2 worker, and 3 worker. The results for the 1 worker scenario are 19.3 requests per second and 332 ms for response time. The results for the 2 worker scenario are 41.4 requests per second and 328 ms for response time. While the results for the 3 worker scenario are 60 requests per second and 331 ms for response time.**

Keywords— **Distributed Task; Python; *RabbitMQ*; Web Crawler; scalability.**

I. PENDAHULUAN

Web crawler atau yang dikenal juga dengan istilah *web spider* atau *web robot* adalah program yang bekerja dengan metode tertentu dan secara otomatis mengumpulkan semua informasi yang ada dalam suatu website. *Web crawler* mengunjungi setiap alamat *website* yang diberikan kepadanya, kemudian menyerap dan menyimpan semua informasi yang terkandung di dalam website tersebut. Setiap kali *web crawler* mengunjungi sebuah *website*, maka dia juga mendata semua link yang ada di halaman yang dikunjungi untuk kemudian dikunjungi lagi satu persatu [1].

Web crawler pada umumnya digunakan untuk mengambil data dari suatu *website* secara otomatis dan mengolah data tersebut sesuai dengan kepentingan yang dieprlukan. Sebagai contoh *web crawler* digunakan untuk mengambil data *review* dari aplikasi yang ada di Play Store dan digunakan sebagai bahan dari data mining [18]. *Web crawler* juga digunakan untuk menghimpun *tweet dari Twitter* yang kemudian data dilakukan *preprocessing text* untuk keperluan *artificial intelligence* [19]. Selain itu, *web crawler* juga berfungsi pengindeksan dari sebuah *website* pada *search engine* dan melakukan generate atau menghasilkan dokumen teks pada *domain* atau *website* tertentu [20].

Pada contoh kasus yang lain *web crawler* digunakan untuk mengambil dan melakukan ekstrak data pada *e-marketplace.* Data digunakan sebagai dasar pengambil keputusan segmentasi pasar dan pengukuran efisiensi. *Web crawler* melakukan penjadwalan untuk mengunjungi setiap halaman kemudian menyimpan data yang telah diekstrak. Data yang disimpan dilakukan agregasi dan dibuat grafik sehingga diperoleh *trend* yang dapat membantu segmentasi pasar [21].

PDC Media Group memiliki *system crawler* produk *marketplace*. *system crawler* ini bertugas untuk mengecek dan membuat matrik perkembangan penjualan dari setiap produk yang telah memiliki penjualan. pada suatu marketplace dilakukan *crawling* dan mencatat produk dan dimasukkan ke dalam suatu *database*. *system crawler* melakukan *crawling* pada produk yang sudah ada dalam setiap *database* setiap harinya. *system crawler* kemudian mencatat dan membuat matrik perubahan penjualan produk setiap hari.

System Crawler pada PDC Media Group diperlukan sebagai *system* untuk mengumpulkan data produk yang selanjutnya akan diolah sebagai bahan *Business Insight* bagi perusahaan. *System crawler* membuat data yang tersebar pada marketplace

publik menjadi satu tempat di *database* sehingga lebih mudah diakses dan dianalisa. Sebagai contoh data terletak pada *marketplace* tokopedia, *system crawler* akan mengunjungi laman produk mengambil data yang diperlukan seperti *view* dan *review* yang kemudian akan dimasukkan ke dalam *database* internal perusahaan. Selain itu, dengan adanya *system crawler* tugas *data entry* menjadi lebih mudah dan sedikit *human error.*

Pertumbuhan dan jumlah data menjadi masalah bagi *system crawler*. PDC Media Group sudah memiliki sekitar 4.500.000 data produk dari online marketplace dengan menyimpan data perubahan transaksi dan order selama 30 hari terakhir. Data produk yang dimiliki terus tumbuh sekitar 10.000 data baru per minggu seiring *crawler* menemukan produk produk baru. Hal tersebut membuat *system crawler* yang sudah ada tidak dapat mengecek semua perubahan transaksi dan order terhadap data produk setiap harinya.

Pada jurnal "*Parallel computational workflows in Python*" dijelaskan infrastruktur *system* / *workflow* program yang memiliki skalabilitas lebih baik. arsitektur yang digunakan adalah *distributed task* dengan menggunakan bahasa python. dimana arsitektur *system* secara umum dibagi menjadi 3 bagian, yaitu unit *publisher task* sebagai *generator task*, *message broker* sebagai koordinasi dan pusat endpoint pertukaran data, dan *node worker* yang bertugas untuk menyelesaikan *task* yang sudah di *generate* [3]. *Worker node* dalam arsitektur ini merupakan kumpulan *server* yang saling terkoneksi dengan *message broker*. arsitektur *system* ini menawarkan skalabilitas yang lebih baik tidak hanya *vertical scaling* tapi juga dapat dilakukan *horizontal scaling*. *horizontal scaling* pada arsitektur ini dapat dilakukan secara tak terbatas dengan menambahkan *worker node* sebagai penyelesai *task*. Dengan demikian ketika data / *task* semakin banyak, waktu eksekusi dapat diatasi dengan cukup menambahkan *worker node*.

Pada journal "Supporting *task* parallelism in Python" arsitektur *system* mirip juga dipakai dalam memproses *general task*. untuk menyelesaikan perhitungan data matematis yang membutuhkan resource besar data dipecah menjadi beberapa *task* dan kemudian masing masing diselesaikan oleh *worker node* yang ada [4].

Distributed task dipilih untuk solusi dari masalah karena proses *crawling* data bisa di distribusikan pada *worker node* sehingga memungkinkan waktu eksekusi yang singkat karena proses terjadi paralel di masing masing *worker node*. Skalabilitas hardware dari *system* yang menggunakan *distributed task* tidak bergantung pada vendor atau penyedia jasa layanan server. Sebagai contoh jika *system* membutuhkan sumber daya 4 vCPU (Virtual Central Processing Unit) dan 8 GB RAM, sementara penyedia layanan server memberikan layanan server paling tinggi dengan spesifikasi 2 vCPU dan 4 GB RAM. Sistem masih bisa scaling dengan 2 *worker node* 2 vCPU dan 4 GB RAM.

Peran *Distributed Task* pada *system crawler* agar *system* memiliki fleksibilitas performa yang dapat mengikuti jumlah data yang ada. Ketika data terus tumbuh, performa *system crawler* dapat terus juga ditingkatkan. Dibandingkan tanpa menggunakan *distributed task*, peningkatan performa hanya dapat dicapai dengan melakukan upgrade spesifikasi server. Sedangkan dengan menggunakan *distributed task* peningkatan performa dapat dicapai baik dari sisi menambah jumlah server atau melakukan upgrade spesifikasi server. Dimana hal tersebut lebih fleksibel karena *system crawler* bukan hanya *scaling resource* secara vertikal tapi juga horizontal.

Penelitian ini bertujuan untuk membuat *system crawler* yang menerapkan arsitektur *distributed task*. Perbedaan *system crawler* yang dibuat dari *system crawler/web crawler* yang sudah ada dan dibahas sebelumnya adalah *system crawler* tidak hanya mampu melakukan ekstraksi data dari sebuah website tetapi juga dapat menyesuaikan *workload* dari data yang terus tumbuh dalam perusahaan. Sehingga ketika *workload crawling* data terus tumbuh, *system crawler* dapat melakukan *scaling* sumber daya dengan mudah. Rumusan masalah dalam penelitian ini adalah bagaimana merancang *system crawler* dengan menerapkan *distributed task* dan bagaimana pengujian performa dari *system crawler.*

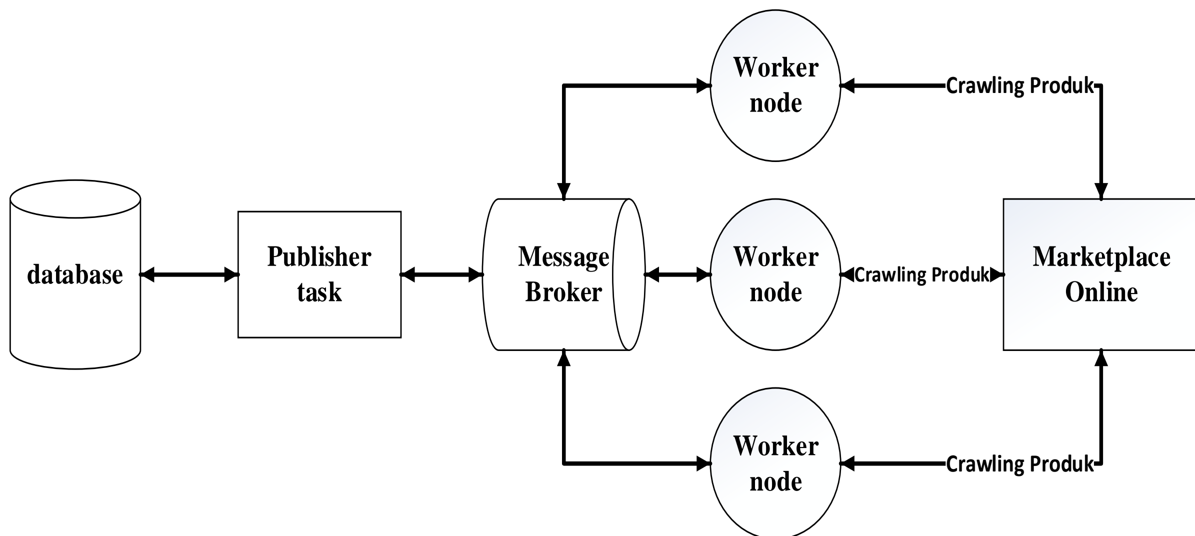
II. METODE

Dalam pembuatan sistem ini digunakan metode pengembangan sistem yaitu metode *waterfall* menurut referensi *Sommerville*, yaitu metode yang menggambarkan proses software development dalam aliran *sequential*. Model *waterfall* yaitu suatu metodologi pengembangan perangkat lunak yang mengusulkan pendekatan kepada perangkat lunak. Sistematik dan sekuensial yang mulai pada tingkat kemajuan sistem pada seluruh analisis, desain, kode, pengujian dan pemeliharaan.[18]

A. Analisis Sistem

Desain *system crawler* dirancang dengan menggunakan *distributed task*. Proses tugas yang paling penting dalam *system crawler* yaitu pada proses *crawling* dan *parsing data* didistribusikan dan di eksekusi di *node worker*. *System crawler* yang dirancang menggunakan *distributed task* dibagi menjadi 3 komponen seperti blok diagram pada Gambar 1.

p-ISSN : 2443-2210
e-ISSN : 2443-2229



Gambar 1. Diagram *system crawler* Distributed *task*

Publish task berfungsi sebagai unit komponen yang memproduksi *task*. *Task* sendiri merupa data *object* yang dibuat oleh *publisher task* berisi data *URL* yang akan di *crawling*. *Publisher task* mengambil data *URL* dari api *database* yang disediakan. *Task* tersebut kemudian di serialisasi kemudian dikirim ke *message broker*. *Publisher task* juga menunggu hasil *task* yang telah dieksekusi oleh *worker node* yang dikirim oleh *message broker* dari *Worker Node* untuk dimasukkan kembali ke *database*.

Message broker digunakan sebagai media komunikasi dan bertukar pesan antara *worker node* dan *publisher task* dalam *crawler system*. *Message broker* akan menerima *task* yang dikirim oleh *publisher task* untuk di distribusikan ke sejumlah *worker node* yang ada. Dalam perancangan ini digunakan *Rabbitmq* sebagai *message broker*.

Worker Node adalah unit komponen yang melakukan eksekusi dari *task* yang telah diberikan oleh *message broker*. Jumlah *Worker Node* dalam suatu *system* bisa lebih dari satu. *Worker Node* terus menunggu *task* dari *message broker*. Ketika *worker node* mendapat *task*, *worker node* melakukan eksekusi dari *task* yang telah didapat. Hasil *task* yang telah selesai dieksekusi dikirim kembali ke *publisher task* melalui *message broker* untuk diolah kembali oleh *publisher task*.

~~B.~~

~~C.~~B. *Flowchart System*

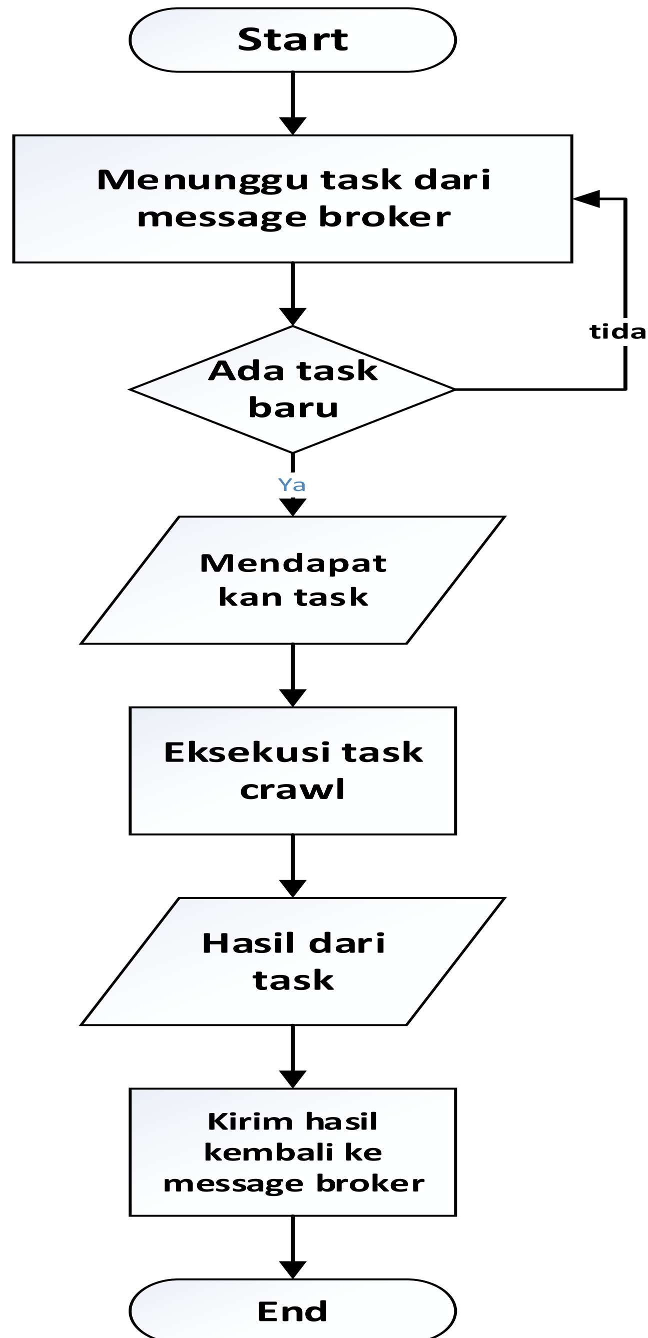
Flowchart system adalah *flowchart* yang menampilkan tahapan atau proses kerja yang sedang berlangsung di dalam sistem secara menyeluruh. Selain itu *flowchart* sistem juga menguraikan urutan dari setiap prosedur yang ada di dalam sistem. Proses *distributed task* dibagi menjadi 2 *flowchart* yaitu *flowchart publisher task* dan *flowchart worker node* yang menjelaskan proses dari kedua komponen yaitu komponen *publisher task* dan komponen *worker node*.

JuTISI
Jurnal Teknik Informatika dan Sistem Informasi

Gambar 2. Flowchart Publisher Task

Pada gambar 2, proses *publisher task* dimulai dengan mengambil data *URL* yang belum dilakukan *crawling*. Setelah data diambil kemudian *publisher task* membuat *task object* dari *URL* yang didapat. Task yang telah dibuat kemudian dikirimkan ke *message broker* untuk dieksekusi oleh *worker node*. Publisher *task* akan menunggu hasil *task* yang sudah selesai dari *message broker*. Hasil dari *task* adalah merupakan data hasil *crawling* yang kemudian disimpan ke dalam *database*.

p-ISSN : 2443-2210
e-ISSN : 2443-2229

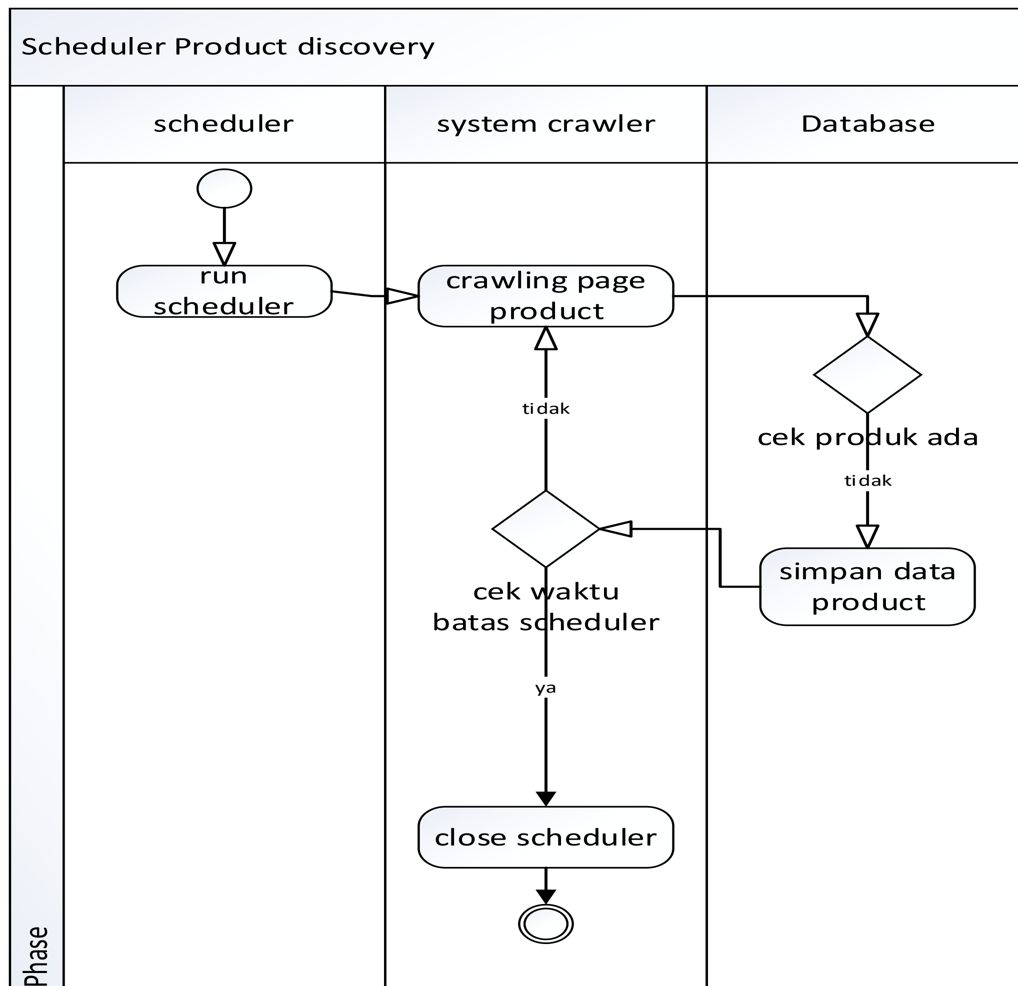


Gambar 3. Flowchart *worker node*

Gambar 3 adalah *flowchart worker node* yang bertugas untuk melakukan eksekusi *crawling task* yang sudah dikirim oleh *publisher task*. Proses dimulai dengan *worker node* melakukan koneksi ke *message broker*. *Worker node* akan mengecek dan menunggu untuk mendapatkan *task* yang dikirim dari *publisher task*. Ketika *worker node* mendapatkan *task*, *task* akan dieksekusi oleh *worker node* dan dilakukan *crawling* dari *URL* yang diberikan *task*. Hasil *crawling* dari *task* yang sudah selesai kemudian dikirimkan kembali ke *message broker* dan akan diolah lebih lanjut oleh *publisher task*.

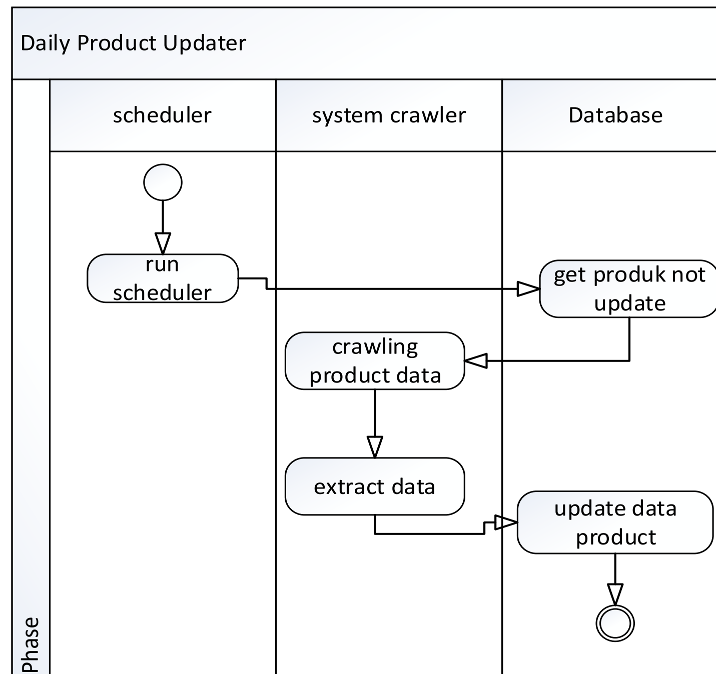
~~D.~~C. *Activity Diagram*

Activity Diagram merupakan rancangan aliran aktivitas atau aliran kerja dalam sebuah sistem yang akan dijalankan. *Activity Diagram* juga digunakan untuk mendefinisikan atau mengelompokan aluran tampilan dari sistem. Pertama adalah *activity diagram Product Discovery*. yang berfungsi untuk menemukan produk baru dalam *system crawler*.



Gambar 4. Activity *Product Discovery*

Pada gambar 4 adalah activity dari *product discovery*. *Product Discovery* dijalankan oleh *scheduler* seperti *cronjob*. Alur pertama *system* akan melakukan *crawling product*. Kemudian *system* melakukan cek produk di *database*. Jika *product* sudah ada maka *product* akan diabaikan. Sebaliknya jika *product* tidak ada maka *product* akan ditambahkan ke dalam *database*. Selain itu sistem juga akan melakukan cek batas waktu *scheduler*. Jika *system* berjalan melewati batas waktu *scheduler* maka *product discovery* akan dimatikan.

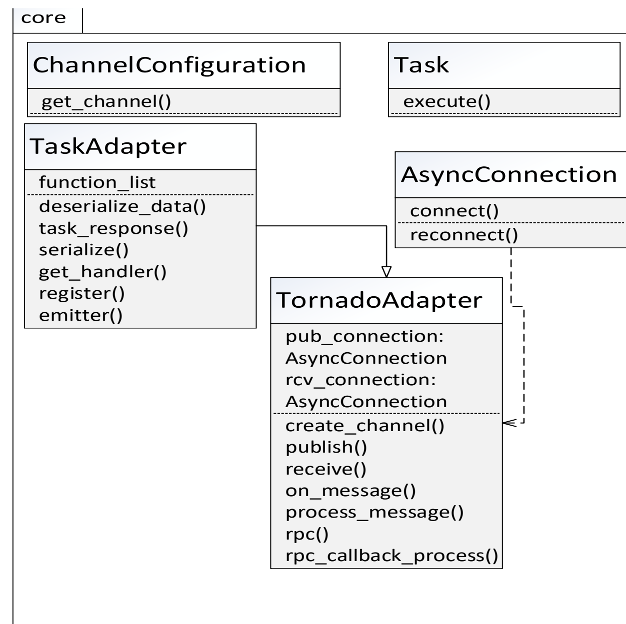


Gambar 5. Activity Daily Product Updater

Pada gambar 5 adalah *activity diagram* dari *daily product updater* sebagai komponen yang melakukan pencatatan perubahan statistic *product* setiap harinya. Daily product updater dijalankan oleh *scheduler* seperti *cronjob*. System akan mengambil semua data *product* dari *database*. Data *product* dilakukan *crawling* kembali oleh *system*. Data baru dari setiap *product* hasil dari *crawling* disimpan kembali di dalam *database*. Selain itu *system* juga akan melakukan cek batas waktu *scheduler*. Jika *system* berjalan melewati batas waktu *scheduler* maka *daily product updater* akan dimatikan.

~~E.~~D. *Class Diagram*

Dalam *system crawler* terdapat beberapa kelas penting yang mengatur dan mengelola *task* yang dieksekusi secara distribusi. Gambar 6 adalah kelas penting dari *system crawler.*



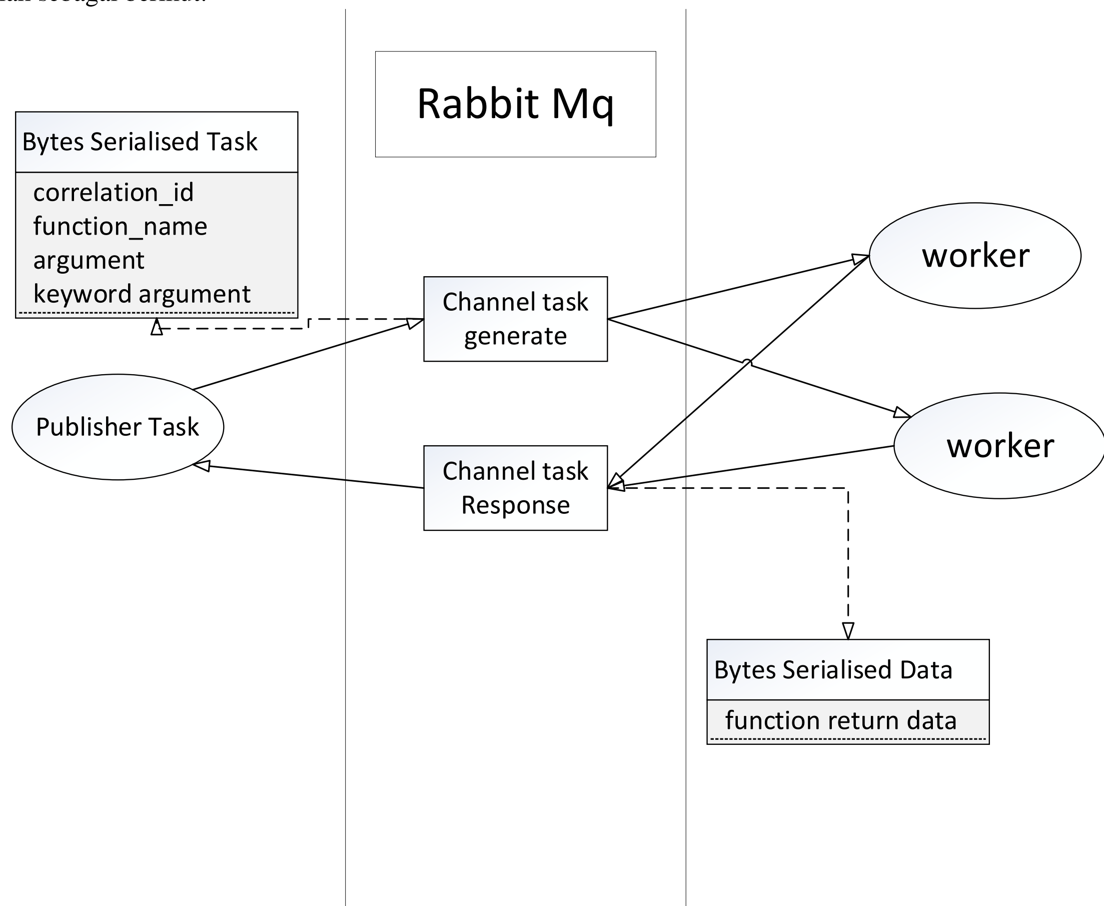
Gambar 6. Class diagram *distributed task*

JuTISI
Jurnal Teknik Informatika dan Sistem Informasi

Kelas *channel configuration* berfungsi sebagai tempat konfigurasi *channel* yang akan digunakan dalam berkomunikasi dengan *RabbitMQ*. Kelas *task adapter* berfungsi mengatur pertukaran data *task*, kebutuhan serialisasi data, deserialisasi data, pengirim dan penerima *task* dari *RabbitMQ*, dan juga sebagai *register* fungsi yang bisa dieksekusi secara *distributed*. Tornado Adapter merupakan *inheritance* dari *Task Adapter* yang berfungsi agar fungsi yang telah *diregister* di dalam *distributed task* dapat juga dijalankan secara *asynchronous*. Task merupakan kelas untuk *object* yang berisi fungsi *distributed* yang telah *diregister* melalui *task adapter*. Task ini yang akan diserialisasi atau deserialisasi dan dieksekusi di *worker*. Kelas *Async Connection* berfungsi untuk membuat koneksi *asynchronous* ke *RabbitMQ*

~~F.~~E. *Desain Eksekusi Task Distributed*

Pada *system crawler* lama eksekusi *crawling* hanya dapat dilakukan di local. Sedangkan pada *system crawler* baru *task* bisa dieksekusi local maupun *distributed* ke *worker node* yang ada. Desain pola komunikasi agar bisa suatu *task* menjadi *distributed* adalah sebagai berikut:



Gambar 7. Pola Komunikasi Distributed

Gambar 7 adalah pola komunikasi *distributed*. Pertama eksekusi *task crawling* dilakukan di *worker node*. Di *crawler system* yang baru, argumen fungsi akan di-*generate* menjadi *task* dan di serialisasi menjadi byte. Task yang sudah di serialisasi dikirim ke *RabbitMQ* yang kemudian dibagikan ke *worker node*. Task dalam bentuk Bytes yang telah dikirim ke *worker node* dideserialisasi kembali agar dapat dibaca *worker*. kemudian *worker* akan melakukan eksekusi dari *task* telah dideserialisasi. Hasil eksekusi *worker node* dari *task* adalah berupa function return data. Function return data ini di-serialisasi kembali dan dikirimkan ke *RabbitMQ* yang selanjutnya akan diterima oleh komponen yang melakukan *generate task* / *publisher task*.

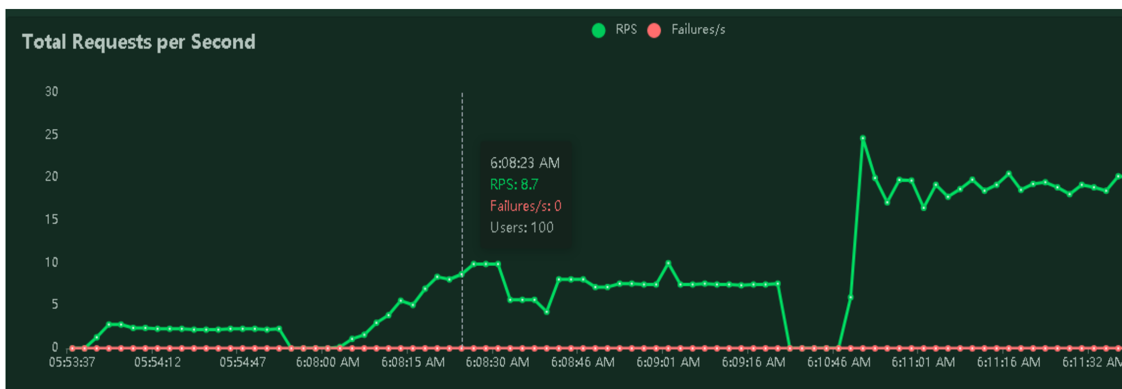
p-ISSN : 2443-2210
e-ISSN : 2443-2229

III. Hasil dan Pembahasan

Hasil dan pembahasan meliputi *load testing crawler system* yang telah dibuat sesuai dengan desain dan juga testing *crawler system* di *production server*. Hasil *load testing* adalah berdasarkan pada 3 skenario dengan perbedaan jumlah *worker node.* 3 skenario tersebut adalah testing *system crawler* dengan 1 *worker*. 2 *worker* dan 3 *worker*. Pengujian menggunakan *docker* sehingga resource server sudah terisolasi dengan baik dan juga untuk pengukuran *RPS* dan *Response Time* menggunakan framework testing yaitu *locust*.

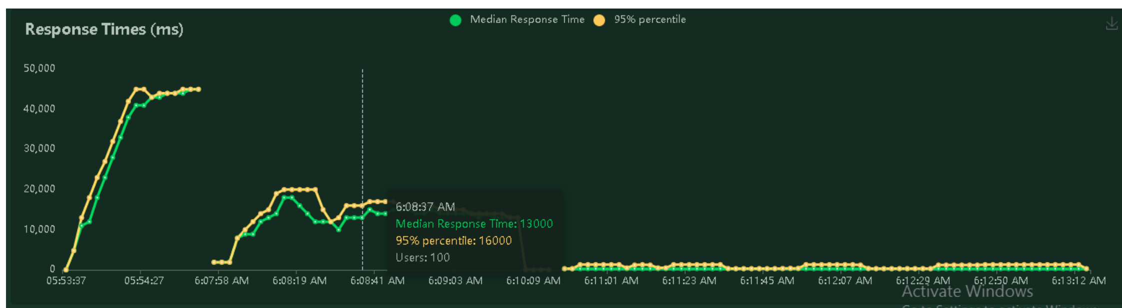
A. Skenario Testing dengan 1 Worker

Untuk testing pertama, *worker* yang dijalankan berjumlah 1 *worker*. *resource server* yang digunakan per *worker* adalah 1 *CPU core* dan dan 512 *MB* memori. Hasil dari testing skenario adalah sebagai berikut:



Gambar 8. Hasil RPS dengan 1 *worker*

Gambar 8 adalah hasil RPS dari *task crawling* yang berhasil dieksekusi oleh *system crawler* dengan 1 *worker*. *Task* yang berhasil dieksekusi oleh *system crawler* berada di kisaran 5 – 25 request per detik.

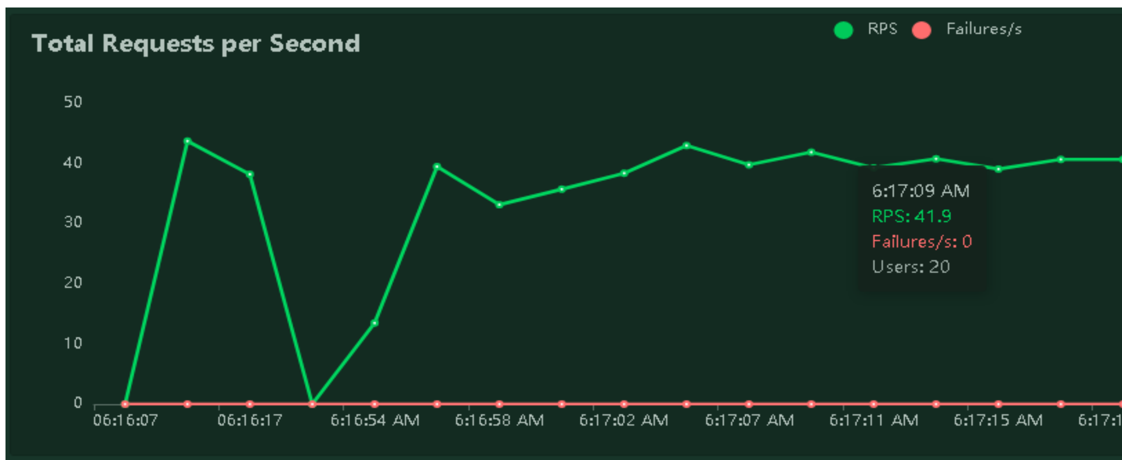


Gambar 9. Hasil Response Time dengan 1 *worker*

Gambar 9 adalah hasil *Response Time*. *Response time* adalah waktu tunggu yang diperlukan oleh *crawler* untuk menyelesaikan suatu *task*.

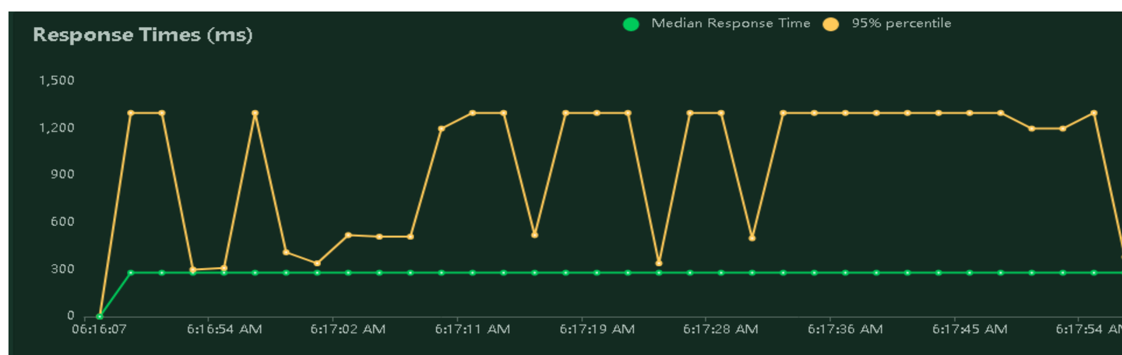
B. Skenario Testing dengan 2 Worker.

Testing kedua adalah mencoba mengukur *system crawler* dengan 2 *worker*. Karena *environment testing* menggunakan *docker*, untuk melipat gandakan *worker* dapat dilakukan dengan perintah "docker compose scale *worker*=2". Jadi sekarang *system crawler* memiliki 2 *worker* dengan masing masing 1 CPU dan 512 MB memori. Hasil dari testing skenario adalah sebagai berikut:



Gambar 10. Hasil RPS dengan 2 *worker*

Gambar 10 menunjukkan report RPS pada locust setelah *scaling worker* menjadi 2 menjadi naik. RPS pada *crawler* berkisar di 40 *request per second*.



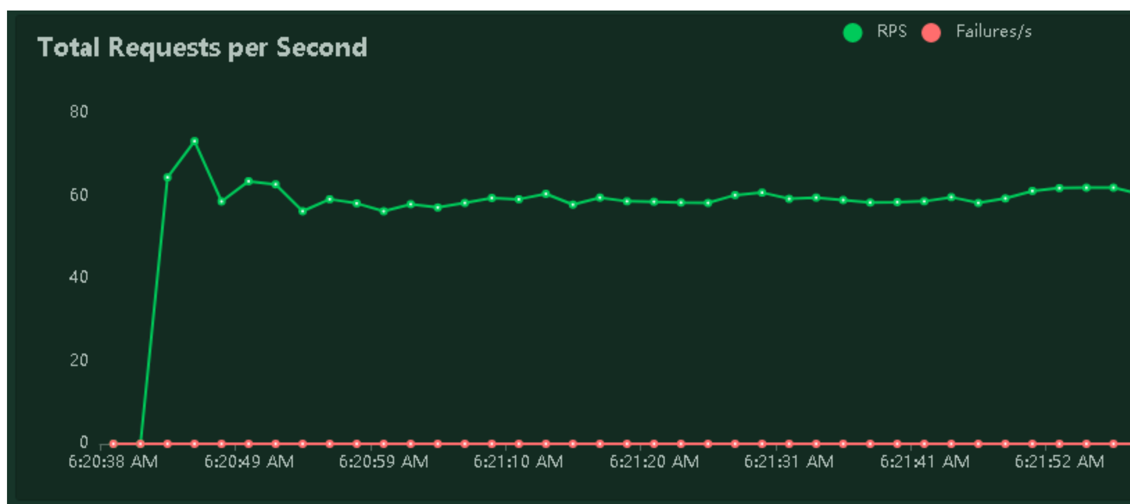
Gambar 11. Hasil Response Time dengan 1 *worker*

Gambar 11 menunjukkan hasil *Response Time*, waktu yang dimiliki *crawler system* berkisar diantara 300 sampai 1.200 *millisecond*.

C. Skenario Testing dengan 3 Worker.

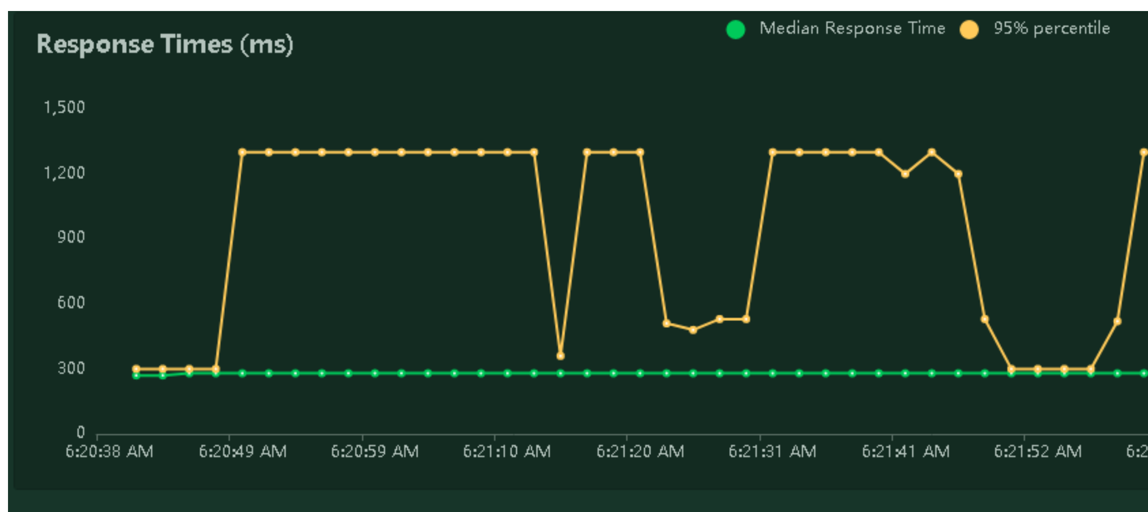
Testing *system crawler* yang ketiga adalah dengan mencobanya dengan 3 *worker*. Untuk *scaling worker* dengan perintah yang sama dengan yang dilakukan sebelumnya yaitu "docker compose scale *worker*=3". Sekarang *system crawler* memiliki 2 *worker* dengan masing masing 1 CPU dan 512 MB memori. Hasil dari testing skenario adalah sebagai berikut:

p-ISSN : 2443-2210
e-ISSN : 2443-2229



Gambar 12. Hasil RPS dengan 2 *worker*

Gambar 12 menunjukkan hasil RPS pada locust setelah *scaling worker* menjadi 3 menjadi naik lebih tinggi dari yang sebelumnya 2 *worker*. RPS pada *system crawler* berkisar di 60 *request per second*.



Gambar 13. Hasil Response Time dengan 3 *worker*

Gambar 13 menunjukkan hasil *Response Time*, waktu yang dimiliki *system crawler* berkisar antara 300 sampai 1.200 ms hampir sama dengan 2 *worker*.

D. Hasil Testing Dari Ketiga Skenario.

Dari hasil testing *system crawler* dengan ketiga skenario didapat data sebagai tabel berikut:

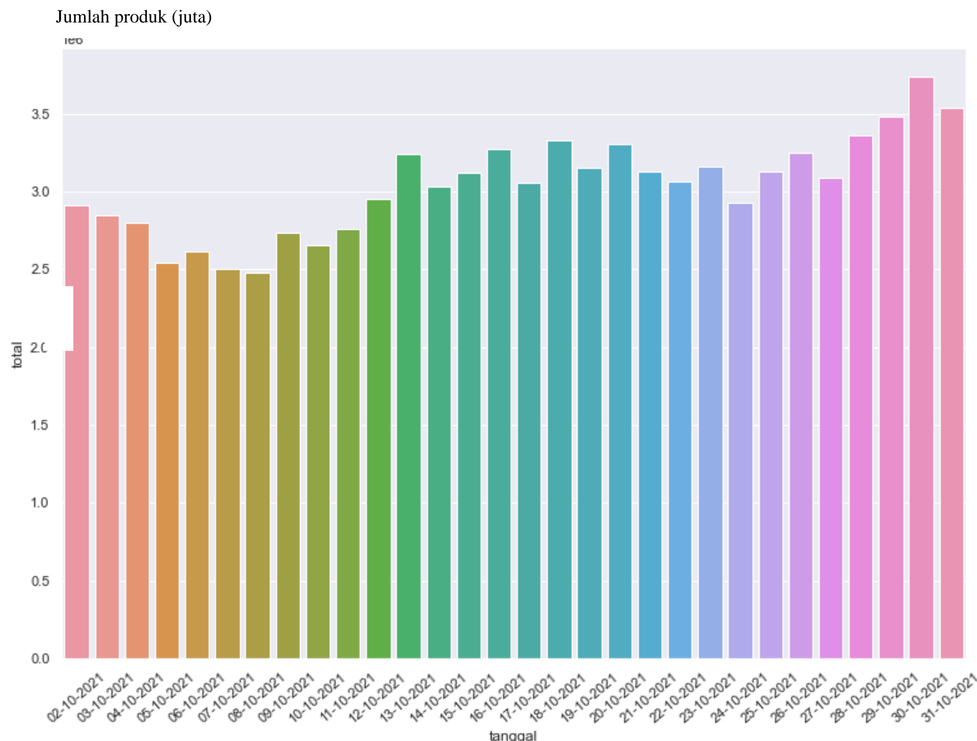
TABEL 1
TABEL HASIL TESTING 3 SKENARIO

Skenario	**RPS (Request Per Second)**	**Response Time**
1 *worker*	19.3 request per second	332 ms
2 *worker*	41.4 request per second	328 ms
3 *worker*	60 request per second	331 ms

Pada tabel hasil testing dapat diperoleh beberapa informasi dari percobaan ketiga skenario. Setiap penambahan *worker* pada *system crawler*, RPS (Request Per Second) ada *system crawler* juga mengalami peningkatan kurang lebih 20 RPS. *Response Time* pada setiap skenario tidak mengalami perubahan yang signifikan.

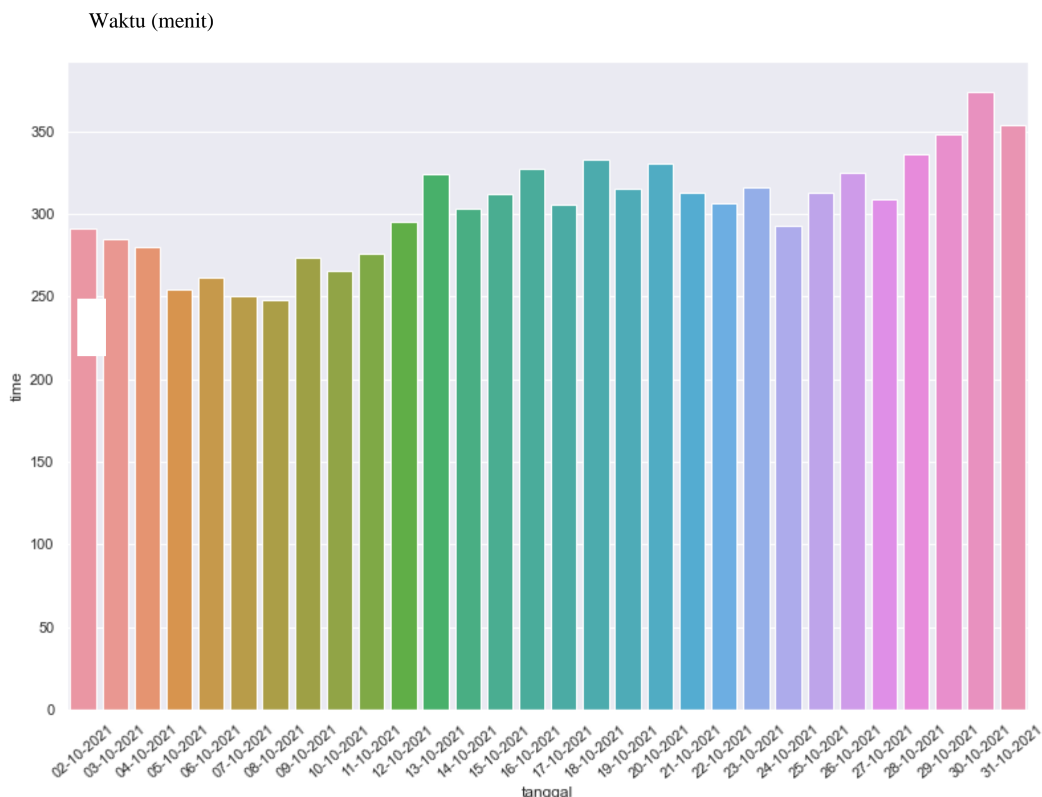
E. Hasil Pengujian Penggunaan Crawler pada Production Server

System crawler juga diuji di *production server*. Pengujian pengguna berfungsi untuk melihat hasil kinerja dari *system crawler* yang sebenarnya di server asli dengan melihat seberapa lama *system* melakukan *crawling* dengan data yang sudah ada di PDC Media Group. Spesifikasi dari *production server* adalah 1 vCPU dan 1 GB untuk masing masing *worker* dan *tasker* dengan jumlah *worker* yang ada yaitu 10 *worker*. Hasil pengujian pertama adalah grafik hasil jumlah produk dari *crawling*.



Gambar 14. Grafik jumlah produk 30 hari terakhir

Gambar 14 merupakan jumlah produk yang berhasil di *crawling* menggunakan *system*. Rentang data yang ada berkisar antara 2.5 juta sampai 4 juta. Data tersebut mengalami penambahan atau pengurangan dalam setiap hari karena ada beberapa data produk yang dihapus dari *web* atau produk baru yang terupload di *web*. Hasil pengujian yang kedua adalah grafik dari waktu yang diperlukan untuk *crawling*.



Gambar 15. Grafik waktu Crawling

Gambar 15 adalah statistik waktu yang diperlukan untuk melakukan *crawling data* yang telah dijelaskan pada gambar 4.30. Dengan jumlah dan pertumbuhan data pada gambar 4.30, juga dicatat waktu yang diperlukan *system crawling* untuk melakukan *crawling* update data yang sudah ada. Waktu yang diperlukan *system crawling* adalah berkisar antara 250 menit sampai 400 menit. Waktu yang dibutuhkan *system* untuk *crawling* berbeda karena dipengaruhi oleh jumlah data yang ada pada gambar grafik sebelumnya.

IV. SIMPULAN

Penerapan *distributed task* pada *system crawler* dapat dicapai dengan membuat *system crawler* menjadi 3 komponen yaitu *publisher*, *message broker* dan *worker*. Penyesuaian performa dan seberapa banyak *system crawler* dapat melakukan *crawling* per detik terhadap jumlah yang ada juga bisa melakukan penambahan *worker* atau upgrade spesifikasi *worker*. *Worker* yang ada tidak harus ada dalam satu mesin yang sama dengan spesifikasi besar. Tetapi juga dapat dipisah ke mesin server yg lebih kecil tapi banyak. System *Crawler* diuji dengan menggunakan framework testing yaitu *locust*. Pengujian dilakukan dengan 3 skenario dengan membedakan jumlah *worker* yang ada di *system*. Dari pengujian *system* yang dilakukan jumlah *request per second* pada *system* meningkat seiring dengan penambahan *worker*. Untuk skenario 1 *worker*, hasil yang didapat adalah 19.3 *request per second* dan 332 ms untuk *response time* yang dibutuhkan. Untuk skenario 2 *worker,* hasil yang didapat adalah 41.4 *request per second* dan 328 ms untuk *response time* yang dibutuhkan. Sedangkan untuk skenario 3 *worker*, hasil yang didapat adalah 60 *request per second* dan 331 ms untuk *response time* yang dibutuhkan. Berdasarkan hasil testing yang dilakukan, performa *system crawler* dapat ditingkatkan dengan melakukan penambahan pada *worker node.* Sehingga kebutuhan *workload system crawler* dapat disesuaikan dengan jumlah *worker node* yang ada. Penyesuaian *worker node* dapat berarti juga *system crawler* memiliki kemudahan dalam melakukan *scaling horizontal* dengan lebih baik. Pengembangan *system crawler* kedepannya adalah melakukan penambahan fitur *reporting* dan *logging* pada *system crawler* sehingga visibilitas dan statistik terhadap data yang telah di *crawling* lebih baik.

DAFTAR PUSTAKA

[1] Wijaya, "Pentingnya Web Crawling sebagai Cara Pengumpulan Data di Era Big Data," 12 7 2017. [Online]. Available: https://www.teknologi-bigdata.com/2016/07/web-crawling-di-era-big-data.html.
[2] E. Tejedor *et al.*, "PyCOMPSs: Parallel computational workflows in Python", *The International Journal of High Performance Computing Applications*, vol 31, no 1, pp. 66–82, 2017.
[3] P. E. Hadjidoukas, A. Bartezzaghi, F. Scheidegger, R. Istrate, C. Bekas, & A. C. I. Malossi, "torcpy: Supporting task parallelism in Python", *SoftwareX*, vol 12, bl 100517, 2020.
[4] R. Mitchell, *Web scraping with Python: Collecting more data from the modern web*. " O'Reilly Media, Inc.", 2018.
[5] A. B. Bondi, "Characteristics of scalability and their impact on performance", in *Proceedings of the 2nd international workshop on Software and performance*, 2000, pp. 195–203.
[6] K. C. Laudon & C. G. Traver, "E-commerce Business Technology Society. 4th", *United State of America: Pearson*, 2008.
[7] R. Anandhi & K. Chitra, "A challenge in improving the consistency of transactions in cloud databases-scalability", *International Journal of Computer Applications*, vol 52, no 2, pp. 12–14, 2012.
[8] E. Åström, "Task Scheduling in Distributed System: Model and prototype". 2016.
[9] M. van Steen & A. S. Tanenbaum, "Distributed system: principles and paradigms", *Pearson*, 2017.
[10] L. Magnoni, "Modern messaging for distributed sytem", in *Journal of Physics: Conference Series*, 2015, vol 608, bl 012038.
[11] K. Apshankar *et al.*, "Web services business strategies and architectures", in *Web Services Business Strategies and Architectures*, Springer, 2002, pp. 1–7.
[12] D. Kuhlman, *A python book: Beginning python, advanced python, and python exercises*. Dave Kuhlman Lutz, 2009.
[13] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming*, vol 11. Addison-Wesley Reading, 2000.
[14] J. O'Hara, "Toward a commodity enterprise middleware: Can AMQP enable a new era in messaging middleware? A look inside standards-based messaging with AMQP", *Queue*, vol 5, no 4, pp. 48–55, 2007.
[15] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, & P. Manzoni, "Testing AMQP protocol on unstable and mobile networks", in *International Conference on Internet and Distributed Computing System*, 2014, pp. 250–260.
[16] M. Yan, H. Sun, X. Liu, T. Deng, & X. Wang, "Delivering Web service load testing as a service with a global cloud", *Concurrency and Computation: Practice and Experience*, vol 27, no 3, pp. 526–545, 2015.
[17] I. Sommerville, "Software processes", *Software Engineering*, pp. 30–31, 2011.
[18] L. B. Ilmawan, "Membangun Web Crawler Berbasis Web Service Untuk Data Crawling Pada Website Google Play Store", *ILKOM Jurnal Ilmiah*, vol 10, no 2, pp. 215–224, 2018.
[19] B. R. Aditya, "Penggunaan Web Crawler Untuk Menghimpun Tweets dengan Metode Pre-Processing Text Mining", *Jurnal Infotel*, vol 7, no 2, pp. 93–100, 2015.
[20] A. Halim, R. D. Nyoto, & N. Safriadi, "Perancangan Aplikasi Web Crawler untuk Menghasilkan Dokumen Teks pada Domain Tertentu", *JUSTIN (Jurnal Sistem Dan Teknologi Informasi)*, vol 5, no 2, pp. 114–117, 2017.
[21] A. Surahman, A. F. Octaniansyah, & D. Darwis, "Teknologi Web Crawler Sebagai Alat Pengembangan Market Segmentasi Untuk Mencapai Keunggulan Bersaing Pada E-Marketplace", *Jurnal Komputer Dan Informatika*, vol 15, no 1, pp. 118–126, 2020.