Fault Tolerance pada Microservice Architecture dengan Circuit Breaker dan Bulkhead Pattern

Edward Hosea, Henry Novianus Palit, Lily Puspa Dewi
Program Studi Informatika Fakultas Teknologi Industri Universitas Kristen Petra
Jl. Siwalankerto 121-131 Surabaya 60236
Telp. (031) – 2983455, Fax. (031) – 8417658
E-mail: edwardhosea1313@gmail.com, hnpalit@petra.ac.id, lily@petra.ac.id

ABSTRAK

Microservice arsitektur sebagai solusi dari sistem arsitektur monolitik semakin berkembang dan banyak digunakan saat ini. Sebagai solusi yang cukup berhasil tidak lepas dari masalah baru yang muncul dalam sistem *microservice*. *Microservice* dikembangkan secara mandiri dan komunikasi antar jaringan sering kali mengalami *fault*. *Fault* yang terjadi dapat disebabkan karena *latency* yang besar atau karena masalah internal yang menyebabkan proses melambat. Hal ini dapat menyebabkan *microservice* tidak dapat diakses lagi oleh user atau *microservice* lain yang membutuhkan.

Penelitian ini memberikan solusi dalam pengimplementasian sistem *microservice* yang dapat tahan terhadap *fault* dengan menerapkan *stability pattern* dengan *circuit breaker* dan *bulkhead pattern*. Pengujian dilakukan pada implementasi sistem *microservice* menggunakan studi kasus sistem PRS (Pendaftaran Rencana Studi) yang digunakan oleh Universitas Kristen Petra. *Fault tolerance* yang diimplementasikan untuk menguji perbandingan performa metode *circuit breaker* DFTM dengan *circuit breaker, bulkhead* dengan tanpa *bulkhead*, serta mengukur tingkat *fault tolerance microservice* dengan MRMM.

Berdasarkan hasil pengujian yang dilakukan didapatkan hasil performa *circuit breaker* DFTM lebih cepat 46% dalam mengembalikan kondisi normal. *Bulkhead* dapat mencegah penggunaan *resource* melebihi kapasitas dari server *microservice* yang dimiliki jika *latency* meninggi. Dari hasil pengujian diketahui bahwa tingkat *fault tolerance* terhadap MRMM menyalahi resiliensi sistem *microservice* diukur terhadap *benchmark*. hal ini dapat dibuktikan dari penurunan tingkat kesuksesan, meningkatnya rata-rata waktu respons, dan penurunan transaksi per detik.

Kata Kunci: *Microservice architecture, circuit breaker, bulkhead pattern*

ABSTRACT

Microservice architecture as a solution to the monolithic architectural system is growing more and more in use today. As a fairly successful solution, the microservice system is not devoid of new problems and other challenges that arise. Microservice as a self-developed system and a service that communicates between networks are often fault. Any fault that occurs can be caused by high latency or because of the internal problem that causes the slowing down process. This may cause microservice to be inaccessible to the user or other microservice in need.

The research provides a solution to the implementation of a microservice system that can withstand fault by applying the stability pattern with a circuit breaker and bulkhead pattern. Tests are made on the implementation of microservice systems using the PRS case study used by the Christian university of Petra. Fault executives are implemented to test the comparative performance of a DFTM method of circuit breaker with a circuit breaker, a bulkhead with no bulkhead, and to measure the fault levels microservice by MRMM.

Based on the results of the tests, DFTM circuit breaker performance was at 46% faster than normal. The results of the testing also have the subtractable use of bulkhead could prevent resource use more than the capacity of microservice servers had if the latency was higher. Testing result also mentioned that the fault tolerance of the microservice measured by MRMM violating resilience of the benchmark. This can be proven from declining success rates, increased average response time, and decreased transactions per second.

Keywords: *Microservice architecture, circuit breaker, bulkhead pattern*

1. PENDAHULUAN

Microservices merupakan arsitektur dalam s*oftware* yang saat ini berkembang pesat untuk mengatasi solusi dari berbagai masalah yang ada pada arsitektur *monolithic* jika sistem tersebut semakin kompleks dan besar. *Microservices* merupakan Arsitektur yang dibangun dengan sistem yang terpisah dan *independent* yang disebut *services*. *Microservices* memiliki tiga keunggulan utama yaitu kecepatan pengiriman, peningkatan *scalability*, dan otonomi yang lebih besar. Pengembangan arsitektur *microservices* masih pada fase awal dan banyak tantangan yang harus dihadapi, salah satunya yaitu sistem yang *fault tolerance* [8]. Faktor utama pendorong pengembangan *microservices yang fault tolerance* yaitu komunikasi antar *services* yang *unreliable* dikarenakan *services* satu dengan yang lainnya berkomunikasi melalui jaringan.

Berdasarkan *fault case* dari industrial survei yang telah dilakukan, salah satu fault yang paling umum terjadi disebabkan oleh *internal faults* dan *service level environment configuration faults*. Penyebab dari dua *fault* tersebut dapat terjadi pada *service* yang memberikan respons salah dan juga adanya kesalahan run*time* yang menyebabkan overload dan denial of *services* [14]. *Fault tolerance* merupakan aspek penting yang perlu dipertimbangkan dan menjadi paling penting karena kegagalan *resources* pada *microservices* dapat mempengaruhi *job*, respons, hasil, dan kinerja sistem [4]. *Fault tolerance* memastikan bahwa sistem dapat merespons dan mengatasi kegagalan yang terjadi pada saat berkomunikasi satu dengan yang lain untuk menjalankan tugasnya. Dengan adanya *fault tolerance*, *fault*/*failure* yang terjadi pada *microservices* dapat diatasi dan sistem tetap tersedia saat dibutuhkan. *Resiliency* suatu sistem s*oftware* sangat diperlukan dan sebagai kebutuhan utama dalam setiap sistem *software* dikarenakan *software* tersebut harus dapat berjalan dan berfungsi setiap saat diperlukan. *Fault tolerance* dengan metode yang telah dikembangkan seperti *Circuit breaker* mampu meningkatkan *stability dan resiliency microservices* yang

mengalami *fault* dan diantisipasi sesuai dengan penyebabnya. *Circuit breaker* dapat mengatasi dampak dari *fault* yang terjadi dengan memonitor target *service* jika *services* tersebut semakin melambat/*unresponsive* ataupun merespons dengan banyak kesalahan. *Circuit breaker* akan memonitor *services* tersebut dan dapat melihat *rate failure* yang terjadi [12]. Metode *Circuit breaker* sendiri masih terus dikembangkan, salah satunya *Dynamic Fault Tolerance Model* dengan mengeliminasi satu *state* dari *Circuit breaker* dan menggunakan algoritma Markov Chain untuk mendeteksi *fault* [1].

Pada penelitian ini *Circuit breaker* dengan DFTM akan menambahkan *pattern* dari DFTM *retry* dan *cache* dengan *bulkhead pattern*. Cara kerja *bulkhead* akan melakukan isolasi *services* yang mengalami *failure* ke dalam *pools*, sehingga *services* lain tetap dapat melanjutkan fungsinya [11]. Skenario *fault*/ *fault case* pada penelitian ini akan menggunakan *environment level* dari *fault case* survei yang dilakukan pada penelitian sebelumnya sesuai dengan mekanisme model *fault tolerance* yang digunakan yaitu *circuit breaker* dan *bulkhead pattern* [14]. *Fault case injection* pada *environment level* yang akan dilakukan pada penelitian ini terbagi menjadi 2 yaitu secara *periodical* akan memberikan *service error response* karena *high load request* untuk menguji *circuit breaker*. *Delay response* untuk menguji *bulkhead pattern*. Tolak ukur yang digunakan dalam pengujian *fault tolerance* pada implementasi *microservice architecture* menggunakan *Microservice Resilience Measurement Model* (MRMM).

2. DASAR TEORI

2.1 *Microservice Architecture*

Banyak perusahaan mulai bermigrasi dari satu *software* arsitektur ke arsitektur lainnya menyesuaikan kebutuhan user yang semakin meningkat, sistem arsitektur yang saat ini berkembang pesat yaitu *microservices* dan pembangunannya masih pada tahap awal [2]. *Microservices architecture* (MSA) dikembangkan berdasarkan filosofi *shared-nothing* dengan struktur sistem *loosely-coupled* yang terisolasi dalam unit yang terpisah [5]. MSA juga sebagai solusi dari sistem arsitektur tradisional yaitu *monolithic architecture* dengan memecah satu aplikasi ke dalam fungsi yang lebih kecil. Setiap fungsi tersebut disebut sebagai *services* yang dapat dikembangkan dan diletakkan terpisah sehingga antar *service* dapat memiliki *language* yang berbeda sehingga memiliki kelebihan utama yaitu lebih mudah di *scalable*.

2.2 *Fault Tolerance*

Aplikasi terdistribusi dengan *microservice architecture* mengharuskan sistem tetap benar, cepat, dan elastis menjadikan tantangan baru dalam penerapannya. Komunikasi melalui jaringan, komponen yang terdistribusi dengan domain yang berbeda, serta elemen yang susah ditentukan dapat menyebabkan *crash/error*. Dengan rentannya sistem terutama komunikasi antar *services* maka dibutuhkan strategi untuk memonitor jalannya *service* dan menggabungkan mekanisme *recovery* agar aplikasi terdistribusi dapat lebih *flexible* dan *robust* [3]. Strategi yang dapat diterapkan yaitu membangun sistem yang toleran terhadap *faults* dengan pemahaman mengenai *fault model*, dengan skenario beberapa *failure* yang mungkin terjadi dan dampak dengan menggunakan parameter jumlah terjadi dan durasi waktu dari skenario tersebut [10]. Dengan menganalisa *fault model* penerapan sistem dapat dilakukan dengan lebih mudah. Pertimbangan dalam membangun sistem yang *fault tolerance* terbagi menjadi empat bagian. Penjelasan model sebagai berikut [6]:

1. *Deal with unavailable service* digunakan dengan tujuan untuk memastikan jika *services* lainnya tidak mengalami *fail* saat *service* yang terpanggil tidak tersedia. *Deal with unavailable* memiliki tiga desain yaitu *fail fast* dimana *service* secepatnya akan mengembalikan *error message*. *Timeout* dapat digunakan untuk memberikan limit pada *service* sampai *service* yang dipanggil melakukan respons.. *Caching for resilient* digunakan saat *services* yang dipanggil *unavailable*, *services* dapat tetap *read/write* data pada *client-side* atau *proxy-side cache*.
2. *Protect services against overload* Dilakukan dengan dua model yaitu *circuit breaker* dimana mekanismenya yaitu untuk mencegah *service* mengalami *failure* dengan menggunakan tiga *state*. Model kedua yaitu *handshaking* menggunakan mekanisme dimana melakukan *handshake* disetiap komunikasi antara yang *request* dan *service* yang menerima *request*. *Service* yang menerima *request* dapat menolak jika telah terjadi *overload*. Kelemahan dari *handshaking* yaitu dibutuhkan *request* dan *respons* extra.
3. *Design for long-term operation* dengan model *steady state*. *Steady state* melakukan mekanisme yaitu membuang data *log* atau entri yang sudah tidak digunakan dalam waktu tertentu..
4. *Localize failure* dengan model *bulkhead*. *Bulkhead* melakukan pembatasan atau isolasi terhadap *service* yang mengalami *failure*. Tujuannya untuk mengurangi dampak dari fault yang terjadi pada *service* dengan memberikan pool untuk masing-masing *service*.

2.3 *Circuit Breaker*

Circuit breaker merupakan salah satu model yang digunakan dalam *fault tolerance* yang berguna untuk mencegah terjadinya *failure* yang disebabkan oleh kegagalan dari satu *services* yang memungkinkan berdampak pada keseluruhan sistem dan dapat menyebabkan down. *Circuit breaker* menggunakan tiga *state* utama yaitu *close*, *open*, dan *half open* [12]. Pada saat *state close request* dapat dilakukan ke target *service*, *faults* yang terjadi akan dicatat dan dimasukkan dalam *circuit breaker counters*. Ketika *counter* yang dicatat telah melampaui syarat yang ditentukan maka *breaker* akan terbuka dan menjadikan ke *state open*. Pada *state open request* yang masuk tidak akan di *passing* ke target *service*, dan akan memberikan *failure message* ke *client* yang melakukan *request*. Pada *state* ini *circuit breaker* dapat melakukan transisi dari *state open* ke *half-open*, dengan secara *periodic* melakukan pemanggilan ke server untuk mengecek apakah server telah *responsive* dalam waktu yang telah ditetapkan. *Half-open state* menerima *request* masuk tetapi tetap dibatasi. Jika target *service* telah merespon hasil *requests* dengan baik maka *circuit breaker* akan direset kembali ke *state close*, dan sebaliknya jika gagal maka akan kembali ke *state open*.

Circuit breaker dengan *Dynamic Fault Tolerance Model* (DFTM) oleh Hajar Hameed Addeen mengeliminasi *state half-open* dari *circuit breaker* dan menggantinya dengan Markov-Chain dengan tiga *state* yaitu *stable*, *unstable*, dan *disable*. Pengeliminasian *circuit breaker* yang diteliti tujuannya yaitu untuk mengganti *timeout* pada saat *detect fault* sehingga lebih menghemat waktu [1]. DFTM yang dikembangkan terbagi menjadi dua level yaitu *upper* dan *lower level*. *Upper level* digunakan untuk mendeteksi *faults* dan terbagi dari tiga *state* markov chain. *Stable state* ketika *microservices available* menerima *request* tanpa ada *fault*. *Unstable state* ketika *microservices* mendeteksi *fault* atau memperbaiki *fault*. *Disable state* ketika *microservices* gagal dalam *recovery* atau *unavailable*. *Lower level* pada DFTM terdiri dari dua *state open* dan *close* yang fungsinya sama dengan *circuit breaker* biasa.

2.4 *Bulkhead Pattern*

Bulkhead pattern membagi *services* ke dalam beberapa *group*, berdasarkan kebutuhan *request*. *Bulkhead pattern* didesain untuk

melakukan isolasi dari *failure* yang terjadi dan membantu agar *services* lainnya yang berbeda *pool* dari *service* yang terisolasi tetap dapat berjalan dan dapat melayani *client* [11]. *Hystrix* sebagai salah satu *library fault tolerance* dari Netflix. *Bulkhead pattern* digunakan untuk mengisolasi titik yang terintegrasi antar *services*, dan menghentikan *failure* antar *services* [9]. *Bulkhead pattern* dapat digunakan dengan dua strategi [7]. Pertama, *semaphores/counters* untuk memberikan limit jumlah panggilan yang bersamaan/*concurrent*. Kedua, yaitu *pool thread* untuk setiap *dependency* akan diberikan masing-masing porsi *thread*. Tujuannya untuk mengisolasi dari pemanggilan *thread* lainnya jika *request* yang sebelumnya masuk menunggu terlalu lama dari target *service* sehingga *pool thread* yang telah diberikan akan diisolasi dan *request* akan meninggalkan *thread* tersebut. *Thread pool* memiliki kelebihan jika *clients libraries* yang dipakai bersifat dinamis.

2.5 *Resilience Benchmark for Fault Tolerance*

Microservices arsitektur membutuhkan analisa dan *debugging* dari sistem tersebut untuk membuktikan bahwa sistem *microservices* yang dibangun benar-benar toleran terhadap *fault*. Namun *research* yang tersedia dalam menganalisa *fault* dan *debugging* sistem *microservices* sebagai tolak ukur/benchmark dari mekanisme *fault tolerance* yang telah digunakan masih sangat terbatas [14]]. Model *benchmark* yang telah dikembangkan untuk mengukur *fault tolerance* pada *microservices* menggunakan *Microservice Resilience Measurement Model*. Konsep model dari MRMM terdiri dari beberapa konsep dari *microservice* sistem yang berhubungan dengan *resilience* yaitu *service, performance, service degradation* untuk mengukur *resilience* mekanisme seperti *load balancing*, *circuit breaker*, API *gateway*, *bulkhead*, dll.

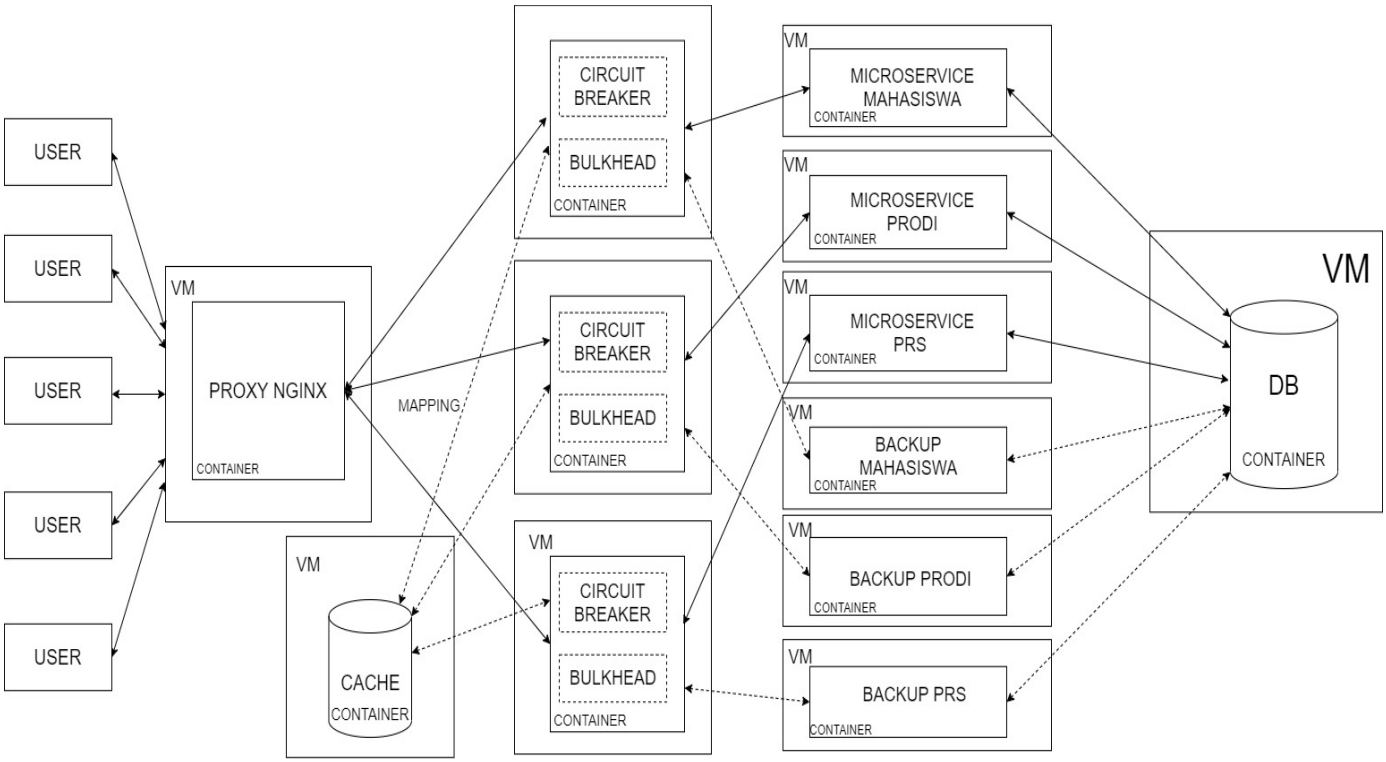
MRMM menentukan *fault tolerance* dengan melakukan pengukuran pada masing-masing *service* dengan parameter sebagai berikut [13]:

1. Pengukuran resiliensi terhadap sistem *microservices* didasarkan pada *service degradation* dengan parameter jumlah *request* yang dapat diterima dalam waktu yang ditentukan. *Service* dapat diukur dengan memberikan batas *performance* rata-rata *service* dan jika *performance* kurang dari batas *performance* tersebut maka dapat dikatakan *degrad*/penurunan resiliensi.
2. Tolak ukur dari sistem *microservice* yang *fault tolerance* diukur dengan 3 perhitungan sebelumnya terhadap 3 *performance attribute* yaitu *response time*, *success rate* dari keseluruhan *service*, dan jumlah *success* per satuan waktu dari *service* sebagai contoh pada studi kasus PRS *service* mata kuliah yang berhasil ditampilkan dari keseluruhan *request* mahasiswa yang masuk.

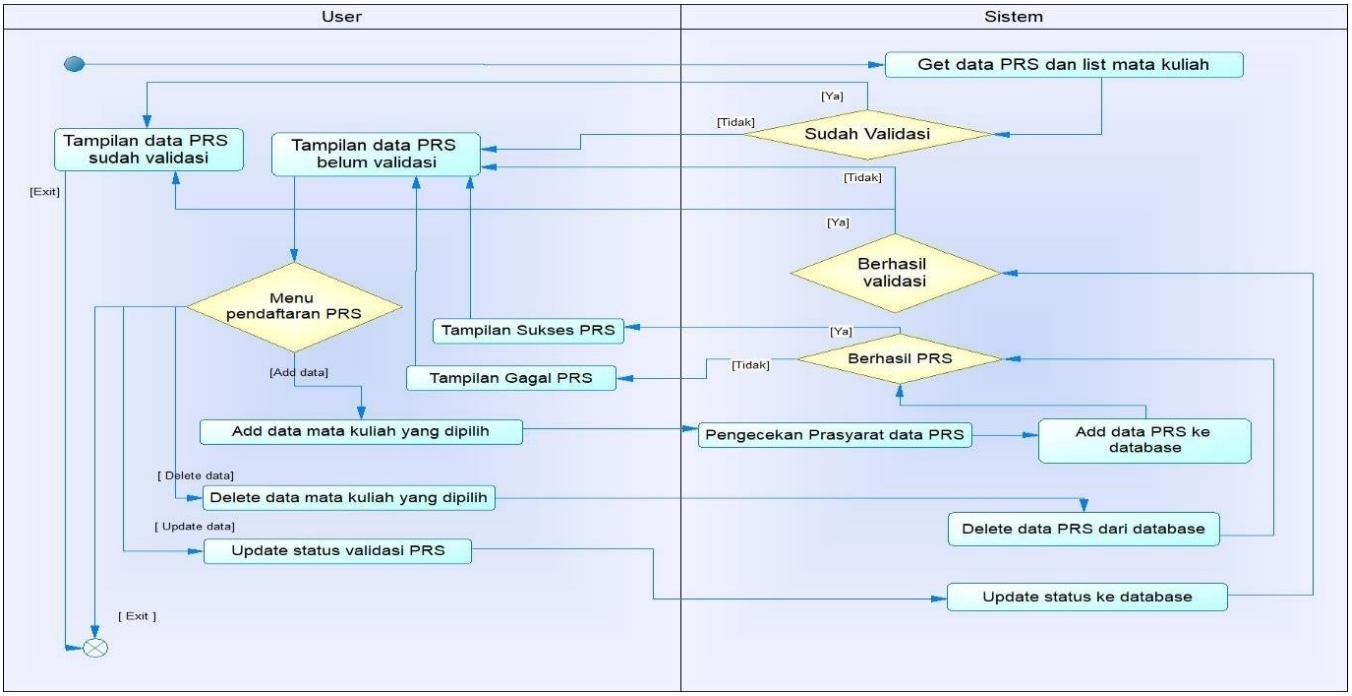
3. DESAIN SISTEM

3.1 Desain Arsitektur Sistem

Desain dari sistem yang diimplementasikan secara keseluruhan diterapkan pada *virtual machine* dengan *cloud* aws. Tiap *virtual machine* menggunakan sistem operasi Ubuntu. Terdapat 6 *virtual machine* digunakan sebagai *microservices* sistem PRS dan *backup microservice* yang diletakkan pada Docker *container*. *Virtual machine* kedua sebagai *gateway* dengan dipasangkan *container* NGINX sebagai penerima *request* dari user yang digunakan untuk pendistribusian *workload* ke masing-masing *service*. *Virtual machine* lainnya digunakan untuk aplikasi metode *fault tolerance* yaitu *circuit breaker* dan *bulkhead pattern* sebagai *endpoint* yang menerima *workload* dari *proxy* NGINX, dimana aplikasi ini yang memonitor dan meneruskan request ke *service* yang menjalankan tugasnya. Berikut pada Gambar 1, dijelaskan desain ar*site*ktur



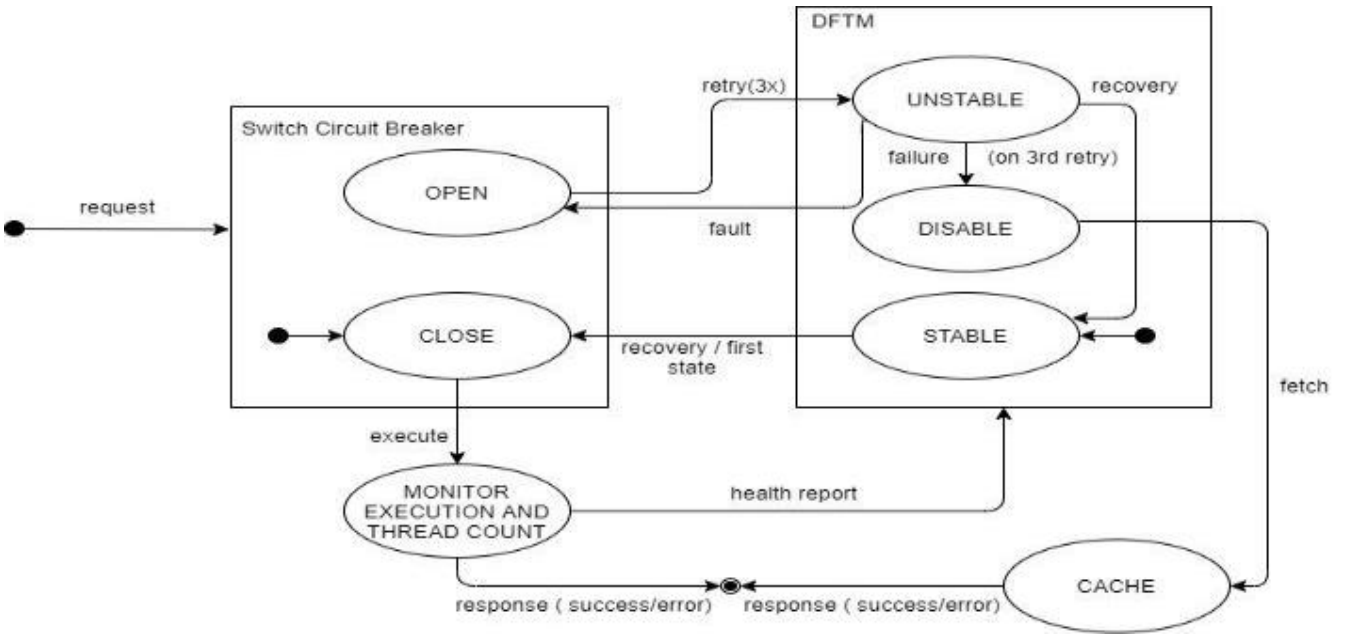
Gambar 1. Desain Arsitektur *Microservices* Sistem PRS



Gambar 2. *State Diagram Circuit Breaker* dengan DFTM

sistem *microservices* yang diimplementasikan dengan *circuit breaker* dan *bulkhead pattern*. *Bulkhead pattern* yang diimplementasikan pada sistem berfungsi untuk membatasi jumlah *thread* agar *request* yang masuk ke dalam server tidak melebihi kapasitas sumber daya yang dimiliki oleh sistem. Sehingga jika *request* melebihi batas *thread* maka akan dilakukan penolakkan *request* oleh sistem.

Detail desain mekanisme *fault tolerance* pada Gambar 2 dengan penjelasan sebagai berikut. *Fault tolerance* menggunakan sebuah fungsi yang digunakan untuk melakukan monitor terhadap *service* target pada sistem *microservices*. *State* awal pada DFTM berada pada posisi *stable* sebagai tanda *service* pada *microservice* aktif akan memberikan event pada *switch circuit* ke *state close*. Pada *switch circuit request* yang masuk dari *client* akan dijalankan pada *state close* dan dicek thread pool jika tidak melebihi batas maka akan dikembalikan *response* ke *client* sesuai dengan *request*nya. Pada saat *response* diberikan juga akan diberikan health *count* ke DFTM, jika *response* yang diberikan *error* maka akan menambah jumlah batas/ threshold pada DFTM. Jika *error* telah melebihi batas maka DFTM akan ke *state* unstable dan memberikan *event* ke



Gambar 3. *Activity Diagram* Proses PRS

switch untuk melakukan perubahan *state* ke open. Untuk setiap *request* yang masuk selanjutnya akan berada pada posisi *open* dan melakukan *retry* ke *service* sebanyak tiga kali, jika tetap gagal maka akan mengembalikan *failure* dan mencoba *cache* sebagai *response* terakhir. Jika *cache* gagal maka akan diberikan *error response*. Jika retry yang dilakukan berhasil DFTM akan beralih ke *state stable* dan akan memberikan *event* ke *switch circuit breaker* untuk ke *state close*, *request* yang masuk selanjutnya akan dieksekusi pada *state close*.

3.2 Desain Alur Aplikasi

Bagian ini menjelaskan mengenai alur sistem aplikasi yang dibuat. Untuk menguji *fault tolerance* dibutuhkan pembuatan sistem *microservices*. Sistem *microservice* yang dibuat menggunakan studi kasus PRS. Implementasi dilakukan pada sistem PRS dengan pembuatan *web service* yang terbagi menjadi 3 *microservice*. Pada Gambar 3 dijabarkan pada *activity diagram* atau sistem alur dari sistem PRS. *Microservice* pada sistem PRS yang diimplementasikan memiliki *dependency* yang tidak terikat satu dengan yang lain dan hanya sebatas pemanggilan *query* dan bukan *publish subscribe*.

4. PENGUJIAN SISTEM

Pengujian dilakukan dengan mengukur kinerja dari *fault tolerance pattern* dari *circuit breaker* dan pengaruh penggunaan *thread pool* dari *bulkhead pattern* terhadap *microservices architecture* serta pengukuran tingkat resiliensi *microservice* pada saat terjadi degradasi dengan *microservice resilience measurement model*.

4.1 Pengujian *Circuit Breaker* dengan DFTM dan tanpa DFTM

pengujian sistem dilakukan dengan melakukan fault *injection* terhadap server PRS yang telah diimplementasikan dengan melakukan failure yaitu *high load exception* dengan melakukan request terhadap microservice mahasiswa dalam proses login. Request dilakukan dengan jumlah 2 concurrent user untuk memenuhi threshold circuit breaker dan melakukan looping request sebanyak infinity. User yang melakukan *request* akan menerima

*error time*out dan jika mencapai threshold *error* yang ditentukan maka *circuit breaker* akan berada pada posisi terbuka dan melakukan *retry*. Berikut pengujian rata-rata waktu *retry* yang dilakukan *circuit* hingga kembali ke *state close* dan dapat menerima *request* kembali seperti semula dengan uji coba sebanyak 10 kali.

Pengukuran performa dari DFTM dan non DFTM diukur dari waktu awal hingga akhir *state* kembali ke posisi *close* dengan evaluasi pada Tabel 1 berdasarkan beberapa hal berikut:

- Waktu eksekusi dari *circuit breaker* DFTM dari waktu circuit terbuka sampai kembali pada *state* recover yaitu *close* dan memberikan *response* success.
- Waktu eksekusi dari *circuit breaker* non DFTM dari waktu circuit terbuka sampai kembali pada *state close* dan kembali memberikan *response* success.
- Pengujian perbandingan circuit breaker DFTM terhadap circuit breaker non DFTM dengan menghentikan proses selama kurang lebih 8 detik dengan pengujian sebanyak 10 kali. Waktu yang dibutuhkan untuk mencapai *state open* selama 5 detik sehingga waktu riil yang dibutuhkan *microservice* kembali sehat adalah 3 detik.
- Kalkulasi waktu rata-rata eksekusi yang dibutuhkan DFTM.
- Kalkulasi waktu rata-rata eksekusi yang dibutuhkan non DFTM.
- Kalkulasi dan perbandingan persentase performa dari DFTM terhadap non DFTM dalam mengabaikan waktu tunggu half-state *circuit breaker*.

Pengujian yang dijabarkan pada tabel dibawah dilakukan dengan melakukan *pause* proses. *Request* yang masuk pertama kali sebagai penentu waktu *state open* kedua metode. Dengan gagalnya request yang melebihi threshold dan terbukanya state open pengukuran dapat dilakukan pada kedua metode. Metode DFTM akan melakukan *retry* dengan eksekusi waktu tertentu, sedangkan non DFTM akan melakukan *sleep time* dengan waktu *timeout* yang diberikan. Dari tabel di atas perhitungan persentase performa antara DFTM dan non DFTM didapatkan bahwa performa DFTM lebih cepat 46.433% dibandingkan non DFTM.

Tabel 1. Hasil Performa *Circuit Breaker* DFTM dan Non DFTM Pause Proses 8 detik

N	DFTM (*second*)	Non DFTM (*second*)
1	4.167	5.316
2	3.213	5.877
3	1.615	5.770
4	1.707	5.081
5	1.606	5.305
6	4.809	5.261
7	3.455	5.311
8	3.398	6.022
9	3.611	5.22
10	1.709	5.52
Average	2.929	5.468

4.2 Pengujian dengan *Bulkhead Pattern* dan tanpa *Bulkhead*

Dari hasil *testing* yang dilakukan dengan *bulkhead pattern* dan tanpa *bulkhead pattern* jumlah penggunaan *resource* tanpa *bulkhead* untuk *microservice* prs dan mahasiswa lebih besar dibandingkan dengan *bulkhead pattern*. Jika *concurrent request* yang masuk lebih besar maka *thread* yang akan terpakai juga akan mengikuti jumlah *request* untuk pemanggilan *service*. Dengan adanya *bulkhead* pemakaian *resource* seperti CPU dan memory dapat dikontrol sehingga server dapat mencegah penggunaan *resource* melebihi kapasitas yang dimiliki server pada saat *fault* terjadi. Hasil *testing* yang telah dilakukan dengan *bulkhead pattern* penggunaan CPU dan *thread* lebih stabil dan tidak melebihi jumlah *thread pool* yang diberikan yaitu maksimum 300 *thread*.

Hasil *testing* yang dilakukan tanpa *bulkhead* dalam menjalankan tugas dari *testing environment* yang telah diberikan tidak dapat diselesaikan dikarenakan *failure* yang diberikan yaitu *delay* atau *latency* yang tinggi menyebabkan penumpukan proses dan menyebabkan aplikasi *bottleneck* pada *memory* yang menyebabkan *container* berhenti dan *worst case* yang didapatkan penggunaan CPU melebihi batas sumber daya sehingga sistem harus di restart manual sebagai bentuk *recovery*. Hasil dari penggunaan CPU hanya dapat dilihat dari *cloudwatch* sebelum server benar-benar tidak dapat menjalankan proses lagi dari waktu terakhir pemakaian CPU hampir melebihi kapasitas.

4.3 Pengukuran Tingkat Resiliensi dengan *Microservice Resilience Measurement Model*

Dengan Perhitungan jumlah transaksi per detik pada saat *microservice* berjalan dengan normal dan saat terjadinya degradasi pada *service* dengan pengukuran *performance attribute* yaitu *Response Time, Success Rate, Success Order* untuk setiap *microservice*. *Benchmark* yang ditetapkan dalam pengukuran resiliensi performa *response time* dan *success rate* yaitu 3 detik dan 90% success rate. *Benchmark* ditetapkan berdasarkan hasil pengujian pada saat *microservice* dalam kondisi sehat. *Testing* yang dilakukan untuk mengukur MRMM berdasarkan *sample testing* pada penjelasan sebelumnya pada proses pengujian.

Perhitungan *performance loss* sulit dilakukan dengan data yang dinamis dan perbedaan sample *request* serta *latency* yang terjadi saat pengujian. Sehingga perhitungan dilakukan berdasarkan perbandingan jumlah *sample* dengan asumsi jumlah *sample* saat sehat dan pada saat terjadi *fault* adalah sama**.** Hasil yang didapatkan untuk pengukuran MRMM untuk *microservice* mahasiswa pada saat terjadi *fault delay*. Hasil dari tabel *Summary Report* digunakan mengukur masing-masing *performance* terhadap *benchmark performance* dengan MRMM dijelaskan sebagai berikut:

- *Recovery rapidity* selama pengujian untuk seluruh *microservices* dengan hasil 1200 milidetik atau sama dengan waktu *delay* selama proses berjalan.
- *Average response* time untuk *microservices* mahasiswa selama proses *fault* terjadi sebesar 5665 milidetik, *microservice prodi* sebesar 11071 milidetik.
- Penurunan *success rate* pada saat terjadi *delay* terhadap *benchmark* sebesar 20.26% pada mahasiswa, 35.99% pada prodi, dan 20.57% pada PRS.
- *Disruption Tolerance* dan *Performance Loss* secara kumulatif tiap service untuk seluruh *microservice* dijabarkan pada Tabel 2.

Tabel 2. Hasil Pengukuran *Disruption Tolerance* dan *Performance Loss Services* Sistem PRS

Microservice (*Service*)	*Disruption Tolerance*	*Performance Loss*
Mahasiswa (Login)	33.5/s	80 *requests*
Mahasiswa (Validasi)	10.6/s	40 *requests*
Prodi (GetAllKelas)	3.3/s	108 *requests*
PRS (AddPRS)	1.2/s	1777 *requests*
PRS (GetDataPRS)	15.7/s	79 *requests*
PRS (DeleteDataPRS)	13.4/s	115 *requests*

5. KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan pengujian yang dilakukan pada sistem, maka dapat disimpulkan sebagai berikut:

- *Circuit Breaker* dengan DFTM memiliki nilai *performance* 46% lebih cepat dalam mengembalikan *switch* ke dalam *state close* dibanding *Circuit Breaker* tanpa DFTM. Sehingga *request* dapat diproses lebih cepat tanpa harus menunggu waktu *half-state* atau *sleep time* dari *switch circuit*.
- *Bulkhead pattern* berpengaruh untuk mencegah penggunaan CPU dan *memory* pada saat terjadi *fault*. Dengan adanya pembatasan *thread* penggunaan CPU dan *memory* dapat tetap stabil dan *request* dapat terpenuhi sesuai dengan jumlah dari strategi *thread* yang diberikan.
- Tingkat *fault tolerance microservice* dengan *circuit breaker* dan *bulkhead pattern* dapat dibuktikan dari suksesnya pengujian terhadap *test case* yang dilakukan dengan kondisi *microservice* yang tetap stabil. Namun terjadi degradasi yang dapat dibuktikan dari 3 *performance* atribut yaitu kenaikan *response time*, penurunan *success rate*, dan penurunan *transaction per second* pada saat proses pengujian dengan *fault* terhadap *benchmark*.

5.2 Saran

Berdasarkan hasil pengujian yang telah dilakukan pada sistem yang telah diimplementasikan, maka terdapat beberapa saran sebagai berikut:

- Penerapan *bulkhead pattern* dapat lebih dioptimalkan untuk isolasi terhadap tiap *service* jika terdapat *dependency* antar *microservice* pada sistem yang ada.
- Pengujian dapat lebih akurat jika *tool* untuk *load test* tidak memakan waktu cukup lama dalam melakukan injeksi variabel sehingga waktu *concurrent request* untuk *sequence task* pada *client* komputer yang berbeda memiliki selisih waktu kerja yang kecil
- Pengujian MRMM masih perlu pengembangan lebih lanjut agar lebih akurat dalam mengukur performa saat degradasi dengan data riil yang dinamis.

6. DAFTAR PUSTAKA

[1] Addeen, Hajar Hameed. 2019. A Dynamic Fault Tolerance Model for Microservices Architecture. *Electronic Theses and Dissertations*.3410.URI=https://openprairie.sd*state*.edu/etd/3410

[2] Baboi, M., Iftene, A. & Gifu, D. 2019. Dynamic Microservices to Create Scalable and Fault Tolerance Architecture. *Procedia Computer Science*, 159, 1035-1044. DOI=https://doi.org/10.116/j.procs.2019.09.271

[3] Cassar, I., Francalanza, A., Mezzina, C., A., & Tuosto, E. 2017. Reliability and Fault-tolerance by Choreographic Design. *Programming Language.* URI=http://arxiv.org/abs/1708.7233v1

[4] Chamoli, S., K., Rana, D. S., Dimri, S., C. 2015. Fault Tolerance and Load Balancing algorithm in Cloud Computing: A survey. *International journal of advance research in computer and communication engineering*, 4(7), 92-96. DOI= 10.17148/IJARCCE.2015.4720.

[5] Ghofrani, J. & Lubke, D. 2018. Challenges of Microservices Architecture: a Survey on the State of the Practice. *ResearchGate*.URI=https://www.researchgate.net/publication/328216639_Challenges_of_Microservices_Architecture_A_Survey_on_the_State_of_the_Practice

[6] Haselbock, S., Weinreich, R. & Buchgeher, G. 2017. Decision Guidance Models for Microservices: Service Discovery and Fault Tolerance. *In Proceedings of Fifth European Conference on the Fifth European Conference on the Engineering of Computer-Based Systems, Larnaca Cyprus*. (pp. 1-10). DOI=https://doi.org/10.1145/3123779.3123804

[7] Jacobs, M. 2017. Hystrix Wiki - How It Works. URI= https://github.com/Netflix/Hystrix/wiki/How-it-Works#Isolation

[8] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, 35(3), 24-35. DOI=10.1109/MS. 2018.2141039.

[9] Livora, Bc.,T. 2016. Fault Tolerance in Microservices. *Brno:Masaryk University, Faculty of Informatics*. URI=https ://is.muni.cz/th/ubkja/masters-thesis.pdf.

[10] Medard, M. & Lumetta, S. 2003. Network Reliability and Fault Tolerance. *ResearchGate*. URI=https://www.research gate.net/publication/227991919_Network_Reliability_and_F ault_Tolerance

[11] Microsoft. 2019. What is Bulkhead Pattern ?.URI=https:// https://docs.microsoft.com/en-us/azure/architecture/patterns /bulkhead

[12] Montesi, F., & Weber, J. 2016. Circuit breakers, discovery, and api gateways in microservices. *Software Engineering, 2018*. URI= https://arxiv.org/abs/1609.05830

[13] Yin, K. & Du, Q. 2020. On Representing Resilience Require ments of Microservice Architecture Systems. *Software Engineer ing*, 2020. URI= https://arxiv.org/pdf/1909.13096 .pdf

[14] Zhou, X., Peng, X., Xie, X., Sun, J., Ji, C., Li, W., & Ding, D. (2018). Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering*, 14(8). DOI= 10.1109/TSE.2018.2887384