Implementasi *Semantic-Aware Model Checker* Untuk Mendeteksi *Deep Bugs* Pada Paxos Protocol Apache Cassandra

Raden Budiarto[1] dan Dikaimin[2]

1) STMIK Jakarta STI&K, Jl. BRI Radio Dalam Jakarta Selatan
2) Universitas Suya, Jl. M.H. Thamrin, Panunggangan Utara, Kota Tangerang, Banten
E-mail : raden@jak-stik.ac.id, dikaiyosh@gmail.com

Abstrak

Pada beberapa tahun terakhir ini, sudah mulai banyak bermunculan sistem berbasis cloud namun untuk menjamin sistem tersebut berjalan dengan baik masih menjadi sebuah tantangan. Salah satu tantangan tersebut adalah diperlukannya sebuah metode untuk menguji dan mengevaluasi sistem berbasis *cloud*. Metode tradisional yang non-exhaustive terbukti tidak dapat secara sempurna mengevaluasi sistem berbasis cloud. Oleh karena itu, dibutuhkan model checker yang lebih efisien. Tujuan dari penelitian ini adalah mengimplementasikan *Semantic-Aware Model Checker* pada Protokol Paxos yang terdapat pada Apache Cassandra. Protokol Paxos merupakan protokol yang digunakan untuk menyelesaikan masalah konsensus pada sistem terdistribusi. Protokol ini cukup rumit sehingga perlu diuji supaya dapat mencegah terjadinya kegagalan yang tidak diinginkan pada Apache Cassandra. Metode penelitian yang digunakan adalah eksplorasi dan eksperimental. Eksplorasi dan penelusuran dilakukan untuk identifikasi *deep bugs* kemudian masing-masing algoritme penelusuran dibandingkan tingkat efisien dan efektivitasnya. Hasil pengujian menunjukkan Metode *Semantic-Aware Model checker* memiliki performa yang baik dalam mereproduksi *deep bugs* dikarenakan metode ini dapat menghindari eksekusi yang tidak diperlukan saat mereproduksi *deep bugs*

Kata Kunci : *Computer Network, Cloud System, Distributed Model, Paxos Protocol, Apache Cassandra*

Pendahuluan

Pada beberapa tahun terakhir ini, terjadi kenaikan popularitas dari aplikasi- aplikasi berbasis sistem terdistribusi berskala besar yang dikenal juga sebagai *cloud systems*). Aplikasi-aplikasi modern banyak yang memakai *cloud systems*, seperti *scale-out storage systems, computing frameworks*, *synchronization services*, dan *cluster management services*. Namun, sebagai sistem yang kompleks, *cloud systems* memiliki tantangan untuk bisa berjalan dengan benar. Sejauh ini, menjamin *reliability* dan *availability* dari sebuah *cloud systems* terbukti masih sangat sulit untuk dilakukan [1].

Untuk mengatasi masalah tersebut, dilakukan uji coba dan evaluasi terhadap *cloud systems*. Metode tradisional seperti *testing* yang merupakan metode dengan pendekatan non-exhaustive terbukti tidak cukup untuk menjamin *reliability* daripada *cloud systems*. Metode *testing* hanya menggunakan sejumlah kasus uji untuk mengevaluasi *cloud systems*. Setiap kasus uji didesain untuk bisa mengarahkan sistem ke berbagai macam keadaan yang dapat menyebabkan *bug*. Namun, kasus uji yang didesain tidak dapat menjamin semua kemungkinan yang dapat terjadi pada sebuah sistem. Oleh karena itu, perlu dikembangkan metode baru yang bersifat *exhaustive*, seperti *model checking* [2].

Pada perkembangan selanjutnya muncul *Distributed System Model Checker* (DMCK) yang menggunakan prinsip model *checking* untuk mengevaluasi dan memverifikasi sebuah *cloud systems*. Terdapat beberapa DMCK yang cukup populer seperti Verisoft, MaceMC, Chess, dan MoDist. DMCK akan mencoba semua kemungkinan eksekusi untuk dapat mengevaluasi semua kemungkinan yang dapat terjadi pada *cloud systems*.

Pada sistem terdistribusi, jumlah kemungkinan pengurutan events yang terjadi akan bertambah secara eksponensial terhadap jumlah events. Dengan jumlah yang sangat banyak tersebut, DMCK membutuhkan waktu yang sangat lama untuk dapat mengevaluasi seluruh kemungkinan pengurutan. Oleh karena itu, DMCK tidak dapat mengevaluasi sebuah *cloud systems* dengan efisien. Untuk mengatasi masalah efisiensi DMCK, para peneliti sebelumnya telah mulai mengembangkan teknik baru untuk mempercepat DMCK [3]. DMCK dipercepat dengan mengeliminasi eksekusi yang tidak diperlukan. Salah satu metode yang telah dikembangkan adalah Semantic-Aware Model Checker.

Semantic-Aware Model Checker (SAMC) merupakan Distributed Model Checker (DMCK) yang menggunakan pengetahuan semantik mengenai *cloud systems* yang ingin diuji untuk mengeliminasi eksekusi yang tidak dibutuhkan. Dengan menggunakan pengetahuan semantik, SAMC dapat lebih cepat menemukan bug yang ada pada *cloud systems*[4]

Apache Cassandra merupakan salah satu *cloud systems* yang cukup populer. Beberapa contoh perusahaan teknologi besar yang menggunakan Apache Cassandra adalah Datastax, Inc., Apigee Corp, CapTech Ventures, Inc., Netflix, dan Synchronous Technologies, Inc. [5]. Namun, sistem yang besar seperti Apache Cassandra juga tidak lepas dari *deep bugs*. Sejak versi 2.0.0, Apache Cassandra juga memperkenalkan fitur baru, yaitu lightweight transactions. Yang dapat mempercepat waktu sebuah transaksi, fitur tersebut hanya dapat diimplementasi dengan menggunakan protokol Paxos.

Protokol Paxos merupakan protokol konsensus yang digunakan pada sistem terdistribusi [6]. Protokol Paxos digunakan untuk menjamin konsistensi data yang tersimpan dalam Apache Cassandra, sehingga setiap proses yang dikirim oleh para klien dapat tersimpan dengan urutan yang sama di setiap mesin dalam cluster yang ada, walaupun ada sejumlah mesin dalam cluster yang mengalami gangguan atau kegagalan. Namun protokol ini tidaklah mudah diimplementasi. *Deep bugs* yang membutuhkan urutan kejadian dan kegagalan yang kompleks bisa terjadi pada protokol ini ketika terjadi kesalahan implementasi. Untuk membangun sebuah sistem terdistribusi yang bebas dari kegagalan, maka implementasi dari protokol ini harus bebas dari kesalahan.

Oleh karena itu, tujuan dari penelitian ini adalah mengimplementasikan *Semantic-Aware Model Checker* pada Paxos Protocol yang terdapat pada Apache Cassandra. Hal ini penting dilakukan untuk dapat mendeteksi *deep bugs* yang ada pada protokol Paxos di Apache Cassandra. *Deep bugs* yang ada dapat direproduksi secara terus menerus dan dipelajari untuk mendapatkan solusi perbaikan terhadap *deep bugs* tersebut. Manfaat yang diharapkan dari penelitian yang dilakukan adalah menunjukkan metode SAMC dapat mereproduksi *deep bugs* pada protokol Paxos di Apache Cassandra dengan lebih cepat dibandingkan metode model checking tradisional yang ada.

Tinjauan Pustaka

Deep Bugs

Deep bugs merupakan bug kompleks yang memerlukan sejumlah urutan kejadian dan kegagalan yang terjadi antara node-node dalam sebuah cluster sistem terdistribusi [7]. *Deep bugs* sangat sulit untuk ditemukan namun memiliki efek yang fatal. *Deep bugs* dapat menyebabkan sistem terdistribusi menjadi tidak konsisten dan mengalami berbagai macam kegagalan.

Distributed Model Checker

Distributed Model Checker (DMCK) merupakan sebuah sistem yang secara formal memverifikasi finite-state sistem yang berjalan bersamaan atau sistem terdistribusi. DMCK bekerja dengan cara mencoba semua kemungkinan urutan events. DMCK akan memaksa sistem target ke dalam kasus ekstrem untuk menghasilkan bugs yang sulit ditemukan.

Terdapat dua jenis pendekatan yang dapat digunakan dalam menggunakan DMCK. Pendekatan pertama adalah secara black-box. Pendekatan black-box merupakan pendekatan yang tidak memerlukan mengetahui pengetahuan spesifik mengenai sistem yang akan diuji [8]. Namun, pendekatan ini memiliki kelemahan, yaitu sangat lambat. Pendekatan ini akan menghasilkan semua kemungkinan urutan pesan yang ada.

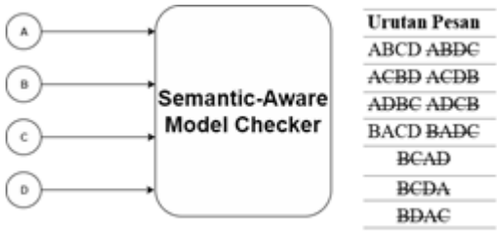
Black-Box Model Checker akan menampung pesan-pesan yang ada terlebih dahulu. Kemudian, Black-Box Model Checker akan

mengacak pesan-pesan tersebut dan membuat urutan pesan-pesan yang akan dieksplorasi. Setelah itu, setiap urutan pesan yang dihasilkan akan dieksekusi dan diuji. Pendekatan ini melakukan permutasi terhadap semua pesan. Sebagai contoh jika ada empat pesan yang diterima maka akan dihasilkan 24 kemungkinan urutan pesan yang harus dieksekusi. Hal ini akan menjadi masalah ketika jumlah pesan yang ada semakin banyak. Dengan semakin banyak jumlah pesan, maka semakin banyak urutan pesan yang harus dieksplorasi dan diuji.

Oleh karena pendekatan black-box sangat tidak efisien maka diperlukan pendekatan lain. Pendekatan yang lain adalah pendekatan *white-box*. Pendekatan *white-box* lebih efisien karena menggunakan pengetahuan tambahan. DMCK yang telah menggunakan pendekatan *white-box* akan mengeliminasi urutan pesan yang perlu dieksekusi. Urutan pesan yang tidak berguna akan dihapus sehingga tidak perlu dieksekusi.

Semantic-Aware Model Checker

Semantic-Aware Model Checker (SAMC) merupakan *white-box Distributed Model Checker* yang memakai pengetahuan semantik bagaimana pesan-pesan diproses oleh sistem target dan menggunakan informasi tersebut pada reduction policies [9]. *Reduction policies* ini merupakan aturan-aturan yang digunakan untuk mengeliminasi pengurutan pesan-pesan yang tidak penting atau tidak perlu dilakukan. Dengan menggunakan reduction policies yang berdasarkan pengetahuan semantik, SAMC dapat mengurangi sejumlah kemungkinan pengurutan events.



Gambar 1: Cara kerja SAMC

Gambar 1 menunjukkan contoh cara kerja SAMC. Ketika SAMC menerima 4 pesan (A, B, C, dan D), maka SAMC akan menyimpan pesan-pesan tersebut dan membuat semua kemungkinan urutan pesan yang mungkin. Dari kemungkinan-kemungkinan yang ada, terdapat beberapa kemungkinan urutan pesan yang jika dieksekusi akan menghasilkan kondisi yang sama pada target sistem. Misalnya, urutan ABCD, ABDC, ACBD, dan ACDB akan menghasilkan kondisi yang sama, maka SAMC tidak perlu mengeksekusi keempat kemungkinan tersebut. SAMC hanya butuh mengeksekusi salah satu dari kemungkinan tersebut. Dengan begitu, SAMC dapat mengurangi banyak kemungkinan urutan pesan yang perlu dieksekusi.

Apache Cassandra

Apache Cassandra merupakan sebuah *distributed database* untuk menyimpan dan mengolah data terstruktur dalam jumlah yang besar yang tersebar di antara server-server [10]. Apache Cassandra didesain untuk menyediakan servis yang memiliki tingkat availability yang tinggi beserta tidak ada *single point of failure*. Apache Cassandra menawarkan kapabilitas yang tidak ditawarkan oleh relational databases dan NoSQL databases yang lain seperti *availability* yang kontinu, performa berskala linear, kesederhanaan operasional, dan kemudahan distribusi data di antara beberapa *data centers* dan *cloud availability zones*.

Apache Cassandra mengimplementasi sistem arsitektur *Peer to Peer*, di mana setiap mesin dalam cluster memiliki peranan yang sama. Pada Apache Cassandra, tidak ada konsep leader ataupun master. Tujuan utama dari Apache Cassandra adalah untuk menyimpan data yang tidak sepenuhnya terstruktur dalam jumlah yang besar dalam bentuk key-value pair.

Salah satu fitur penting dari Apache Cassandra adalah *Lightweight Transactions*. *Lightweight transactions* digunakan untuk memastikan transaction isolation level, yaitu ketika ada dua operasi yang dilakukan oleh dua klien yang berbeda terhadap data yang sama, hanya ada satu urutan modifikasi yang akan tersimpan. Lightweight transactions ini hanya dapat diimplementasi di Apache Cassandra menggunakan protokol Paxos.

Paxos Protocol

Protokol Paxos merupakan protokol yang digunakan untuk menyelesaikan masalah konsensus dalam sebuah jaringan [11]. Masalah konsensus adalah masalah di mana terdapat beberapa proses-proses dalam sebuah sistem yang ingin menyetujui sebuah nilai di mana media komunikasi antara proses-proses tersebut dapat mengalami gangguan dan kegagalan. Masalah ini perlu diselesaikan ketika ada beberapa node yang saling berkomunikasi dalam sebuah *cluste*r dan mencoba untuk melakukan sebuah modifikasi pada data yang sama, protokol Paxos menjamin bahwa hanya akan ada satu nilai yang bakal disetujui semua node atau semua node setuju bahwa tidak ada nilai yang disetujui ketika cluster mengalami gangguan. Protokol Paxos dibutuhkan untuk membangun sistem terdistribusi yang bisa menjamin konsistensi data.

Dalam protokol Paxos, setiap proses akan memiliki peran masing-masing. Terdapat tiga jenis peran dalam protokol Paxos, yaitu proposer, acceptor, dan learner. Fungsi dari setiap peran adalah sebagai berikut:

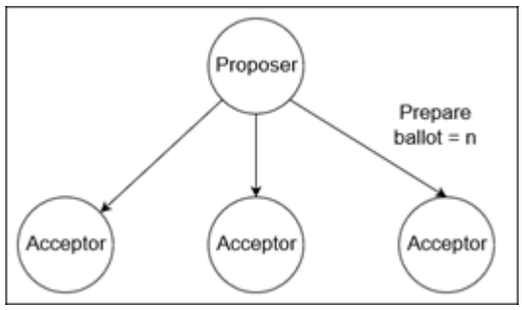
1. *Proposer.* *Proposer* merupakan proses yang berperan untuk mengajukan proposal dengan sebuah nilai.

2. *Acceptor.* *Acceptor* merupakan proses yang berperan untuk menyetujui proposal yang diajukan oleh proposer. Dalam sebuah sistem yang terdiri atas beberapa acceptor, terdapat istilah quorum. Quorum merupakan kumpulan dari beberapa *acceptor* yang jumlahnya mayoritas dari total *acceptor* yang ada pada sistem.

3. *Learner.* *Learner* merupakan proses yang berperan untuk mempelajari nilai yang telah disetujui. Nilai yang telah disetujui didapatkan ketika quorum sudah setuju dengan sebuah nilai.

Proses protokol Paxos dalam memilih sebuah nilai dapat dibagi menjadi tiga tahapan utama. Ketiga tahapan tersebut adalah Prepare, Propose, dan Commit. Secara lebih spesifik proses yang terjadi pada setiap tahapan adalah sebagai berikut [12].

1. Tahapan *Prepare* Pada tahapan ini, *proposer* akan memilih sebuah bilangan unik n yang disebut sebagai ballot. Nilai dari ballot ditentukan berdasarkan waktu dalam sistem sehingga nilai dari ballot yang dihasilkan akan selalu lebih besar dibandingkan nilai dari ballot yang dihasilkan sebelumnya. Setelah menentukan nilai (n), *proposer* akan mengirimkan pesan prepare dengan ballot n kepada semua acceptor.



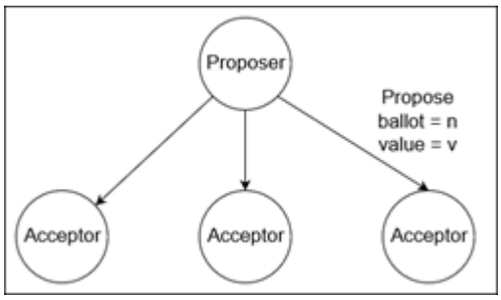
Gambar 2: Tahapan *prepare*

Ketika *acceptor* menerima sebuah pesan prepare dengan ballot n, maka ia akan memeriksa apakah ia telah menerima pesan prepare yang lain dengan nilai ballot n yang lebih besar. Jika ia sudah menerima pesan prepare dengan n yang lebih besar, maka ia akan mengabaikan pesan prepare tersebut. Namun jika belum, maka ia akan berjanji (promise) kepada pesan prepare tersebut untuk selanjutnya tidak menerima pesan prepare dengan ballot lebih kecil daripada n. Setelah itu, *acceptor* juga akan membalas pesan ini ke *proposer* dengan disertakan nilai v yang merupakan nilai dari proposal dengan ballot terbesar yang sudah pernah diterima sebelumnya. Jika *acceptor* belum pernah menerima proposal sama sekali, maka ia akan mengembalikan nilai null.

2. Tahapan *Propose* Tahapan ini dimasuki setelah *proposer* menerima balasan daripada pesan prepare yang telah dikirimkan sebelumnya kepada semua acceptor. Jika mayoritas *acceptor* membalas dengan pesan promise, maka *proposer* akan melanjutkan untuk mengajukan sebuah nilai. *Proposer* akan memilih nilai v dari semua balasan yang ada dengan ballot n terbesar. Kemudian, *proposer* akan mengajukan sebuah proposal dengan nilai v dan ballot n kepada semua acceptor.

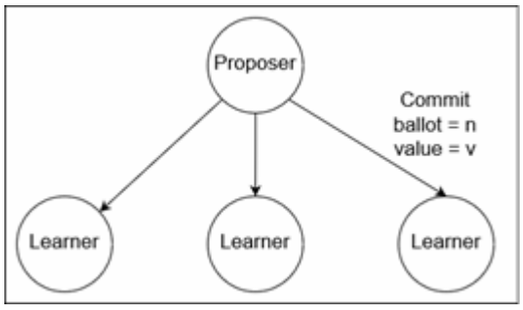
Ketika *acceptor* menerima proposal dengan nilai v dan ballot n, maka ia akan memeriksa apakah ia telah menyetujui (accept) proposal lain dengan ballot yang lebih besar dari n. Jika

belum, maka ia akan menyetujui proposal ini. Setelah menyetujui proposal ini, *acceptor* akan membalas ke *proposer* dengan pesan accept.



Gambar 3: Tahapan *propose*

3. Tahapan Commit Tahapan ini dimasuki setelah *proposer* menerima pesan balasan dari para *acceptor* terhadap pesan propose yang telah dikirimkan sebelumnya. Jika mayoritas dari *acceptor* menyetujui propose tersebut, maka *proposer* akan mulai mengirimkan pesan commit dengan nilai v kepada para learner.



Gambar 4: Tahapan *commit*

Ketika *learner* menerima pesan *commit*, maka ia akan menyimpan nilai tersebut. Kemudian, *learner* akan membalas dengan pesan acknowledge untuk menandakan bahwa ia telah mengetahui nilai tersebut.

Metode Penelitian

Alur Penelitian Rancang bangun sistem dan eksperimen dilakukan dengan beberapa fase sebagai berikut.

1. Studi Pustaka Studi pustaka dilakukan untuk mempelajari berbagai publikasi yang telah dilakukan sebelumnya oleh peneliti-peneliti lain mengenai Distributed Model Checker. Objek studi pustaka juga ditelaah terhadap protokol Paxos dan Apache Cassandra untuk dapat mengimplementasi metode SAMC. Selain itu, juga dilakukan studi terhadap implementasi protokol Paxos di Apache Cassandra. Hal ini dilakukan supaya dapat mendapatkan gambaran reduction policies yang dapat digunakan pada *Semantic-Aware Model Checker*.

2. Perancangan Sistem Sistem testing yang dirancang berupa model checker berdasarkan framework *Semantic-Aware Model Checker*. Sistem dirancang untuk dapat menjalankan tugas-tugas tertentu ke Apache Cassandra. Pada fase ini, juga dirancang reduction policies berdasarkan hasil studi yang telah didapatkan pada fase sebelumnya. Dari sini akan dihasilkan aturan-aturan yang mengikuti framework dari *Semantic-Aware Model Checker*.

3. Pengembangan Sistem Pada tahap ini sistem uji coba dan dibangun dan diintegrasikan dengan Apache Cassandra. Integrasi framework *Semantic-Aware Model Checker* dengan Apache Cassandra diperlukan agar dapat digunakan untuk melakukan uji coba pada Apache Cassandra. Beberapa bagian penting yang perlu diimplementasi pada fase ini adalah:

   (a) Mengimplementasi class-class dari framework SAMC yang diperlukan untuk mengintegrasi SAMC dengan Apache Cassandra.

   (b) Modifikasi kode Apache Cassandra untuk dapat berkomunikasi dengan SAMC.

4. Pengujian Sistem Pengujian sistem dilakukan untuk uji coba mereproduksi *deep bugs* yang ada pada protokol Paxos di Apache Cassandra.*Deep bugs* yang ada perlu direproduksi ulang supaya mudah untuk dievaluasi.

5. Evaluasi Sistem Proses pada tahap ini adalah mengevaluasi jumlah eksekusi yang diperlukan model checker untuk

mereproduksi *deep bugs* pada protokol Paxos di Apache Cassandra. Berikutnya akan dilakukan eksperimen untuk mendeteksi *deep bugs* pada protokol Paxos di Apache Cassandra. Terdapat dua mode untuk menjalankan sistem yang akan dibangun, yaitu:

(a) Mode Eksplorasi. Mode ini digunakan untuk mengeksplorasi dan menemukan *deep bugs*. Terdapat 4 jenis strategi yang dapat digunakan, yaitu:

i. *Random*
ii. *Depth-First Search*
iii. *Dynamic Partial-Order Reduction*
iv. *Semantic-Aware*

Mode eksplorasi juga dapat dijalankan dengan initial path. Initial-path berisikan urutan dari beberapa event yang harus dieksekusi terlebih dahulu. Dengan begitu, SAMC akan mengeksekusi event di initial path terlebih dahulu sebelum kemudian mulai mengeksplorasi dan menentukan event selanjutnya berdasarkan strategi yang digunakan.

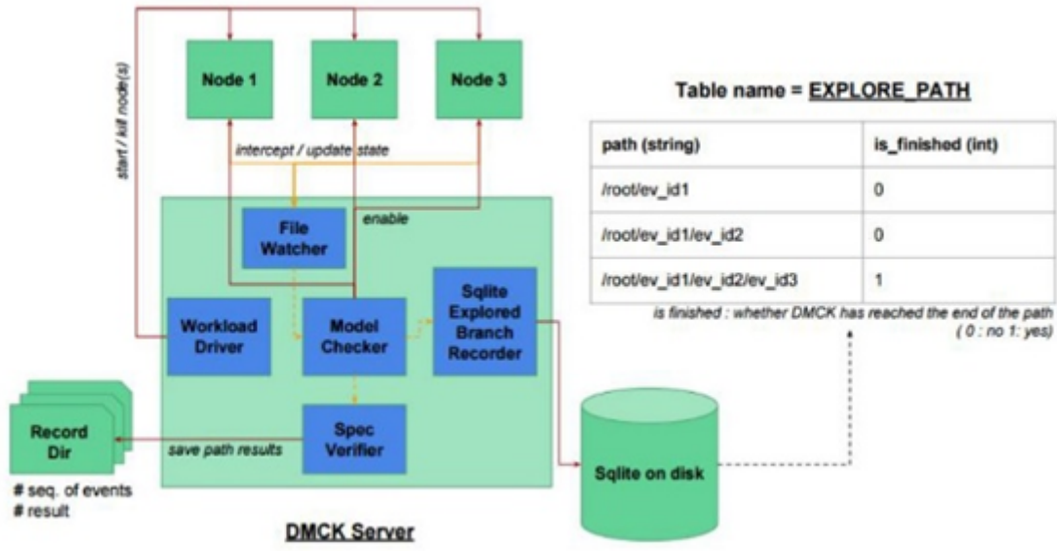
(b) Mode *Programmable*. Mode ini digunakan untuk mengeksekusi sebuah path dengan urutan event tertentu yang sudah tersimpan dalam sebuah file. Tujuan dari metode ini adalah untuk mengulang eksekusi dari sebuah rekaman event yang menghasilkan bug. Mode ini juga dibutuhkan untuk menganalisis sebuah bug. Kemampuan untuk mengulang dan mereproduksi ulang sebuah bug juga dibutuhkan dalam proses debugging.

Perancangan Sistem

Proses menjalankan SAMC dimulai dengan menginisialisasi sistem di working directory. Script inisialisasi akan membuat sebuah working directory bersama file-file lain yang dibutuhkan untuk menjalankan eksperimen. Setelah itu, File Watcher akan diaktifkan. File Watcher merupakan proses yang bertanggung jawab untuk memantau dan mencegah pesan dari setiap node. Setelah SAMC sudah siap, Workload Driver akan memulai untuk melakukan eksplorasi untuk menemukan *deep bugs*. Workload Driver akan memulai sebuah cluster dari Apache Cassandra. Setelah semua node sudah aktif, node-node tersebut akan mulai untuk saling berkomunikasi. Workload Driver kemudian akan mencoba melakukan beberapa lightweight operations. Operasi-operasi ini dilakukan dengan mengeksekusi beberapa Cassandra Query Language (CQL) statement yang menggunakan driver Apache Cassandra. Lightweight operations menggunakan protokol Paxos sehingga akan terjadi pengiriman messages protokol Paxos. SAMC kemudian akan mencegat messages tersebut sebelum dieksekusi oleh setiap node. Pencegatan ini akan menghentikan operasi yang ingin dilakukan node sampai ketika SAMC mengizinkan node untuk mengeksekusi messages yang tercegah tersebut.

SAMC akan menunggu hingga semua events tercegah. Setiap events yang dicegah akan disimpan dalam sebuah queue. Setelah itu, SAMC akan memulai untuk menyusun ulang events yang ada di dalam queue. SAMC kemudian akan memilih event selanjutnya yang akan dieksekusi berdasarkan konfigurasi strategi eksplorasi. Setelah menentukan event mana yang akan dieksekusi, SAMC akan memanggil fungsi *commit*() untuk memerintahkan node yang menunggu untuk melanjutkan eksekusi event yang tercegah sebelumnya.

Setelah semua event yang tercegah telah dieksekusi, SAMC akan memverifikasi keadaan global daripada cluster dan kemudian menaruh hasil eksekusi ke dalam sebuah folder record. Hasil eksekusi akan disimpan di file dengan nama berdasarkan urutan eksekusi. Spec Verifier adalah proses yang bertanggung jawab untuk melakukan verifikasi terhadap hasil eksekusi. Spec Verifier yang akan menentukan apakah terdapat bug atau tidak. SAMC juga akan menyimpan path yang sudah dieksplorasi ke dalam SQLite database untuk menandakan bahwa path tersebut telah dieksplorasi. Penyimpanan ini diatur oleh SQLite ExploredBranchRecorder. Record dalam basis data digunakan untuk menghindari pengeksekusian path yang sama oleh SAMC.



Gambar 5: Perancangan Sistem

Spesifikasi Sistem

Pembangunan sistem dan eksperimen dilakukan dengan menggunakan perangkat dengan spesifikasi berikut. Spesifikasi perangkat keras yang digunakan untuk membangun sistem dan melakukan eksperimen adalah sebagai berikut.

1. Processor: 2.4 GHz Intel Core i7
2. Memory: 16 GB
3. Hard Disk: 500 GB

Sedangkan perangkat lunak yang digunakan untuk membangun sistem dan melakukan eksperimen adalah sebagai berikut.

1. Java versi 1.8.0
2. Text editor Atom versi 1.8.0
3. Apache Ant versi 1.9.9
4. Python versi 3.6.1
5. Apache Cassandra versi 2.0.0

Hasil dan Pembahasan

Implementasi Sistem

Sistem model checker dibangun berdasarkan framework *Semantic-Aware Model Checker*. Sistem ditulis menggunakan bahasa Java. Apache Ant digunakan sebagai build management tool. Skrip-skrip eksperimen yang akan digunakan ditulis dalam bahasa python dan juga bash. Skrip-skrip tersebut akan menjalankan proses program Java sesuai dengan tugasnya masing-masing.

Tahapan pertama adalah membuat Working Directory yang berfungsi sebagai lokasi di mana proses uji coba dilakukan. Proses eksperimen akan dilakukan sepenuhnya pada direktori ini. Direktori "cassandra-2.0.0" dibuat sebagai Working Directory untuk melakukan uji coba terhadap sistem Apache Cassandra versi 2.0.0. Dalam direktori ini, terdapat beberapa file konfigurasi dan Skrip. File- file dijabarkan pada tabel 1.

Tabel 1: Konfigurasi File Skrip

Nama Skrip	Fungsi
dmck.conf	File ini berfungsi sebagai konfigurasi umum untuk sistem uji coba.
target-sys.conf	File ini berfungsi sebagai konfigurasi spesifik untuk Apache Cassandra versi 2.0.0
cassRunner.sh	Skrip ini digunakan untuk memulai proses uji coba.

Experiment-ModeRunner.sh	Skrip ini digunakan untuk melakukan proses uji coba yang diulang berkali-kali dengan jumlah event dalam initial path yang berbeda-beda.
startNode.sh	Skrip ini digunakan untuk menghidupkan node dari Apache Cassandra.
killNode.sh	Skrip ini digunakan untuk menghentikan node dari Apache Cassandra
startWorkload.sh	Skrip ini digunakan untuk menjalankan workload pada Apache Cassandra.
stopWorkload.sh	Skrip ini digunakan untuk menghentikan workload pada Apache Cassandra.
checkConsistency.sh	Skrip ini digunakan untuk memverifikasi apakah ada kesalahan data pada cluster yang sedang diuji coba.
mc_log.properties	File ini digunakan untuk melakukan konfigurasi log daripada model checker.
cass_log.properties	File ini digunakan untuk melakukan konfigurasi log daripada Apache Cassandra.
readconfig	Skrip ini digunakan untuk konfigurasi Skrip.
setup	Skrip ini digunakan untuk membuat salinan daripada working directory yang akan menjadi tempat melakukan uji coba.
Expected-ResultPaths	Direktori ini berisikan hasil akhir yang diharapkan ketika bug terjadi.
initialPaths	Direktori ini berisikan direktori initialpath yang dibutuhkan untuk mereproduksi bug.

Pada tahapan selanjutnya adalah mengimplementasi Java Class yang dibutuhkan Framework SAMC. Class yang perlu diimplementasi supaya SAMC dapat digunakan untuk uji coba pada Apache Cassandra. Beberapa class yang perlu diimplementasi ditampilkan pada tabel 2.

Tabel 2: Konfigurasi File Skrip

Nama Class	**Fungsi**
CassRunner	Class ini berfungsi untuk menjalankan sistem uji coba dan menjalankan cluster Apache Cassandra. Class ini akan melakukan looping untuk melakukan uji coba berkali-kali.
CassSAMC .	Class ini berisikan implementasi reduction policies yang telah dirancang untuk Apache Cassandra. Class ini diwariskan dari class DporModelChecker.java
CassVerifier	Class ini berfungsi untuk melakukan pengecekan terhadap Apache Cassandra apakah terdapat bug terhadap sistem.
CassWorkload Driver	Class ini berfungsi untuk mengatur proses menjalankan workload berupa perintah-perintah yang akan dikirimkan kepada Apache Cassandra.

Pada sisi kode Apache Cassandra, ditambahkan kode Workload.java yang berfungsi sebagai representasi dari perintah yang berhubungan dengan protokol Paxos yang akan diajukan ke sistem Apache Cassandra. Ketika workload ini dijalankan, maka sistem Apache Cassandra akan memulai operasi yang membutuhkan protokol Paxos sehingga terjadi pertukaran pesan protokol Paxos. Selain itu, ditambahkan juga Interception-Layer.java yang digunakan untuk menuliskan data-data yang dicegat ke dalam sebuah file yang akan dibaca model checker. Kode ini juga berfungsi untuk membaca file yang akan ditulis oleh model checker yang berisi perintah-perintah dari model checker.

Pesan protokol Paxos antar node pada cluster Apache Cassandra akan dicegat untuk dapat diacak. Proses pengiriman dan penerimaan pesan-pesan tersebut diatur oleh class Messaging-Service.java pada Apache Cassandra. Oleh karena itu perlu dilakukan modifikasi pada file tersebut.

Dari Gambar 6, dapat dilihat bahwa semua pesan yang merupakan bagian dari pesan protokol Paxos akan ditahan dan dicegat dalam sebuah thread baru. Thread ini akan menyimpan pesan tersebut dalam sebuah paket yang

akan ditulis ke dalam sebuah file. File ini akan dibaca oleh pihak *model checker*. Thread ini akan menunggu sampai diizinkan oleh model checker untuk akhirnya mengirim pesan tersebut.

```
01. public void sendOneWay(MessageOut message, int id, InetAddress to)
02. {
03.     if (message.verb == Verb.PAXOS_PREPARE
04.         || message.verb == Verb.PAXOS_PROPOSE
05.         || message.verb == Verb.PAXOS_COMMIT
06.         || message.verb == Verb.READ
07.         || message.verb == Verb.PAXOS_PREPARE_RESPONSE
08.         || message.verb == Verb.PAXOS_PROPOSE_RESPONSE
09.         || message.verb == Verb.PAXOS_COMMIT_RESPONSE
10.         || message.verb == Verb.READ_REQUEST_RESPONSE)
11.     {
12.         logger.info("[SAMC] interceptMessage: " + FBUtilities.getBroadcastAddress() +
13.                 " sending " + message.verb + " to " + id + "@" + to);
14.         Thread interceptorThread = new Thread(new InterceptorThread(message, id, to));
15.         interceptorThread.start();
16.     } else {
17.         sendMessage(message, id, to);
18.     }
19. }
```

Gambar 6: Potongan kode pencegatan pesan protokol Paxos

Hasil Uji Coba Sistem

Proses uji coba dilakukan dalam dua tahap, yaitu menentukan *initial path* dan mereproduksi *bug*. *Initial path* ditentukan dengan menggunakan mode programmable. Pesan-pesan antara node diacak sedemikian rupa sehingga menyerupai skenario yang diharapkan dapat menyebabkan *bug* terjadi. Berikut initial path yang didapatkan untuk kedua *bug* yang berhasil terdeteksi. *Bug* ini dikenal dengan CASSANDRA-6013 dan CASSANDRA-6023.

CASSANDRA-6013 memiliki tingkat kedalaman sebesar 40. Hal tersebut berarti jumlah pesan yang ada dalam initial path untuk mereproduksi *bug* CASSANDRA-6013 ada 40 pesan. Berikut urutan pengacakan pesan yang dapat menyebabkan terjadinya *bug* CASSANDRA-6013.

CASSANDRA-6023 memiliki tingkat kedalaman sebesar 58. Hal tersebut berarti jumlah pesan yang ada dalam initial path untuk mereproduksi *bug* CASSANDRA-6023 ada 58 pesan. Berikut urutan pengacakan pesan yang dapat menyebabkan terjadinya *bug* CASSANDRA-6023. Proses mereproduksi *bug* dilakukan berkali-kali dengan strategi berbeda-beda. Proses ini dilakukan juga dengan keadaan di mana terdapat jumlah event yang berbeda-beda pada initial path.

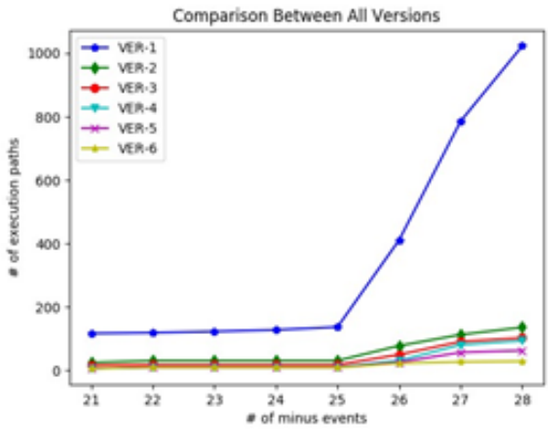
Eksperimen dilakukan pertama kali dengan menggunakan strategi DPOR. Hasil dari strategi DPOR kemudian dianalisis untuk mengetahui pasangan pesan mana yang paling banyak diacak dan merupakan pengacakan yang tidak berguna. Berdasarkan analisa tersebut, dihasilkan *reduction policy* yang baru. *Reduction policy* kemudian diterapkan pada metode SAMC dan dinamakan VER-1 (versi 1).

Selanjutnya proses yang sama diulang terus menerus di mana SAMC VER-1 akan dianalisis untuk menghasilkan SAMC VER-2. Hal yang sama diulang terus menerus untuk meningkat performa SAMC. Setelah melakukan berbagai eksperimen, didapatkan 6 jenis reduction policy. Hasil eksperimen tersebut terdapat pada Tabel 3.

Tabel 3: Hasil uji coba reproduksi *bug*

Min.event	DP-OR	V1	V2	V3	V4	V5	V6
27	1676	786	113	91	81	57	27
26	921	411	78	50	31	26	23
25	457	137	31	18	11	10	9
24	430	128	31	18	11	10	9
23	349	123	31	18	11	10	9
22	189	119	31	18	11	10	9
21	175	117	25	16	6	6	6

Dari Tabel 3 terlihat bahwa terjadi peningkatan performa untuk setiap penambahan *reduction policy*. Hal ini menunjukkan bahwa setiap *reduction policy* dapat mengurangi pengacakan pesan-pesan yang tidak berguna. Hasil perbandingan keenam versi ini ditunjukkan oleh gambar 7.



Gambar 7: Perbandingan antara keenam versi SAMC

Melalui uji coba yang telah dilakukan, didapatkan enam *reduction policy*. *Reduction policy* tersebut dapat dikategorikan berdasarkan jenis pola aturan yang digunakan. Aturan-aturan ini hanya akan berlaku pada protokol Paxos di Apache Cassandra. Berikut *reduction policy* yang diperoleh:

1. *Increment-1* Aturan ini berlaku untuk pasangan pesan yang memiliki tipe pesan yang sama dan berupa balasan terhadap prepare, propose, atau *commit*. Jika pasangan pesan ini memiliki isi yang sama, maka tidak perlu dilakukan pengacakan.

2. *No Conflict Read/Write* Aturan ini berlaku ketika salah satu dari pesan merupakan balasan terhadap *commit*, pesan read, atau balasan terhadap read. Jika terdapat salah satu pesan yang berupa jenis itu, maka tidak perlu dilakukan pengacakan.

3. *Discard-1* Aturan ini berlaku untuk pasangan pesan yang memiliki tipe pesan yang sama dan berupa pesan prepare, propose, atau *commit*. Jika pasangan pesan ini memiliki isi yang sama, maka tidak perlu dilakukan pengacakan.

4. *Discard-2* Aturan ini berlaku untuk pasangan pesan yang sama-sama merupakan balasan terhadap pesan prepare. Jika salah satu pesan merupakan pesan yang menolak prepare, maka tidak perlu dilakukan pengacakan.

5. *Disjoint Update* Aturan ini berlaku untuk pasangan pesan yang berupa pesan prepare dan balasan terhadap prepare. Kedua pesan ini tidak perlu diacak karena tidak akan menghasilkan kondisi yang baru. Hal yang sama juga berlaku untuk pasangan pesan propose dan balasan terhadap propose, dan juga pasangan pesan *commit* dan balasan terhadap propose.

6. *Increment-2* Aturan ini berlaku terhadap pasangan pesan yang sama-sama merupakan balasan terhadap propose. Pasangan pesan ini tidak perlu diacak karena tidak akan menghasilkan kondisi yang baru. Hasil dari implementasi kode program *reduction policy* ini ditampilkan pada gambar 8.

```
1. // Increment
2. if (isPaxosResponse(e1Verb) && e1Verb.equals(e2Verb)
3. && (e1.getToId() == e2.getToId()) && hasSameContent(e1Payload
4. , e2Payload)) {
5.         return false;
6. }
7.  // No conflict R/W
8. if (e1Verb.equals("PAXOS_COMMIT_RESPONSE")
9.         || e2Verb.equals("PAXOS_COMMIT_RESPONSE")
10.        && (e1.getToId() == e2.getToId())) {
11.        return false;
12. }
13. if (e1Verb.equals("READ")
14.        || e2Verb.equals("READ")
15.        && (e1.getToId() == e2.getToId())) {
16.        return false;
17. }
18. if (e1Verb.equals("READ_REQUEST_RESPONSE")
19.        || e2Verb.equals("READ_REQUEST_RESPONSE")
20.        && (e1.getToId() == e2.getToId())) {
21.        return false;
22. }
23. // Discard-1 & 2
24. if (isPaxosCommand(e1Verb) && e1Verb.equals(e2Verb)
25.        && (e1.getToId() == e2.getToId()) && (e1Key == e2Key) &&
26. 23llot.compareTo(e2Ballot) > 0)) {
27.        return false;
28. }
29. 2if (e1Verb.equals("PAXOS_PREPARE_RESPONSE")
30. && (e1.getToId() == e2.getToId()) && (e1Key == e2Key)) {
31. String e1Response = e1Payload.get("response");
32. String e2Response = e2Payload.get("response");
33. if (e1Response.equals("false") || e2Response.equals("false")) {
34. return false;
35. }
36. }
37. // Increment-2
38.  if (e1Verb.equals("PAXOS_PROPOSE_RESPONSE")
39. && e2Verb.equals("PAXOS_PROPOSE_RESPONSE")
40. && (e1.getToId() == e2.getToId())) {
41. return false;
42. }
```

Gambar 8: Implementasi dari *reduction policy*

Pembahasan

Berdasarkan data yang didapatkan dari hasil eksperimen, dapat dilihat bahwa terjadi pengurangan jumlah eksekusi yang diperlukan untuk mereproduksi *bug* tersebut. Dapat dilihat terdapat peningkatan performa dari DPOR ke SAMC VER-1.

Peningkatan performa terjadi karena SAMC VER-1 dibangun berdasarkan metode DPOR. Perbedaannya adalah SAMC VER-1 melakukan eliminasi terhadap pengacakan pesan-pesan yang tidak berguna. Hal tersebut menyebabkan SAMC VER-1 dapat mereproduksi *bug* jauh lebih cepat dibandingkan DPOR.

Peningkatan performa ini hanya dapat dicapai selama reductionpolicy yang diterapkan hanya membuang pengacakan yang tidak berguna yang berarti jika tidak acak akan menghasilkan hasil yang sama. Peningkatan performa juga dialami ketika membuat SAMC

VER-2. SAMC VER-2 menggunakan *reduction policy* yang sama dengan SAMC VER-1. Perbedaannya adalah terdapat penambahan *reduction policy* baru yang didapatkan dari hasil analisa terhadap pengacakan yang tidak berguna pada SAMC VER-1. Dengan begitu performa dari SAMC VER-2 menjadi lebih baik. Hal yang sama juga terjadi sampai pada SAMC VER-6.

Penutup

Berdasarkan penelitian yang telah dilakukan, simpulan yang diperoleh dijabarkan sebagai berikut. Tujuan dari penelitian ini sudah tercapai dengan diimplementasikan metode *Semantic-Aware Model Checker* (SAMC) dan telah teruji saat mendeteksi *deep bug*s dengan menggunakan 6 *reduction policy* yang telah dijabarkan.

Berdasar hasil pengujian pemanfaatan keenam *reduction policy* tersebut berhasil meningkatkan efisiensi urutan pesan hingga 62 kali lipat. Deep *bug*s dapat direproduksi dalam sebuah sistem dengan mengacak pesan-pesan yang terkirim dalam sebuah sistem secara spesifik. Urutan secara spesifik tersebut akan menghasilkan *bug* yang dapat mengacaukan cara kerja sistem. Jenis-jenis *bug* ini sulit direproduksi sehingga diperlukan sistem uji coba yang bersifat otomatis seperti metode SAMC. Dengan demikian dapat disimpulkan metode SAMC dapat menghindari eksekusi yang siasia dalam mereproduksi *deep bug*s. Adapun saran yang dapat diberikan dijabarkan sebagai berikut. Dalam mereproduksi sebuah deep *bug*s pada *cloud systems*, diperlukan analisa terhadap sistem target. Dengan mendapatkan informasi mengenai protokol dari sistem target, maka dapat dihasilkan aturan-aturan yang dapat digunakan untuk pengacakan pesan yang tidak berguna.

Daftar Pustaka

[1] R. Guerraoui & M. Yabandeh, “Model checking a networked system without the network”, Proceedings of the 8th USENIX conference on Networked systems design and implementation, p.28-36, 2011

[2] V. Kumar, A. Agarwal, "HT-Paxos: High Throughput State-Machine Replication Protocol for Large Clustered Data Centers", The Scientific World Journal, p.489-502, vol. 15, 2015.

[3] J. Simsa, R. Bryant, & G. Gibson, “dBug: Systematic Testing of Unmodified Distributed and Multi-Threaded Systems”, SPIN Journal of cloud system, vol. 17, p. 46-57, 2016.

[4] T. Leesatapornwongsa, M. Hao, P. Joshi, J.L. Lukman & H.S. Gunawi, “SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems”, Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation”, p.345-361, 2014.

[5] Anonim, “Companies using Apache Cassandra.” https://idatalabs.com/tech/products/apache-cassandra, 17 Juni 2017

[6] S. Lu, S. Park, E. Seo, & Y. Zhou, ” Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics”., Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, p.329-339, 2015.

[7] T. Leesatapornwongsa, J.L. Lukman, S. Lu, & H.S. Gunawi, “TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems”, Proceedings of the 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, p.304-318, 2016.

[8] R. Banabic, & G. Candea, “Fast black-box testing of system recovery code”, Proceedings of the 7th ACM european conference on Computer Systems, p.202-212, 2012.

[9] H.S. Gunawi, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K.J. Eliazar, and A.D. Satria, “What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems” Proceedings of the 5th ACM Symposium on Cloud Computing, p.289-301 2014.

[10] A. Lakshman, & P. Malik, “Cassandra - A Decentralized Structured Storage System”, International Journal on Large Scale Distributed Systems and Middleware, vol. 34, p.287-301, 2016.

[11] C. Flanagan, & P. Godefroid, “Dynamic Partial-Order Reduction for Model Checking Software”, Journal ACM Principles of Programming Languages, vol. 16, p. 321-334 2015.

[12] R. Budiarto, "Manajemen Risiko Keamanan Sistem Informasi Menggunakan Metode FMEA Dan ISO 27001 Pada Organisasi XYZ", Journal of Computer Engineering, System and Science, vol. 2, p. 48-58, 2017.