출처: https://www.atmarkit.co.jp/ait/articles/2002/10/news038.html

게이머를 위한 무료 음성 텍스트 채팅 응용 프로그램 "Discord '를 개발, 제공하는 Discord는 2020년 2월 5일 (미국 시간) 애플리케이션 기반 서비스의 하나 인 'Read States"를 Rust 언어로 다시 구현하였고, 그 결과 서비스의 성능이 크게 향상 됐다고 공식 블로그에서 밝혔다.

Read States 서비스는 지금까지 Go 언어로 구현되어 있었다. 그럼에도 불구하고 왜 Read States을 Rust로 다시 구현하려고 했는지, 어떻게 다시 구현했는지, 다시 구현하여 어떻게 성능이 향상 되었는지를 설명하였다.
https://blog.discordapp.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f

Rust로 다시 구현한 배경은?

Read States 서비스의 목적은 Discord 사용자가 어떤 채널의 어떤 메시지를 읽었는지를 추적하는 것이다. 즉, 사용자가 Discord에 연결하거나 메시지를 보내거나 메시지를 읽을 때마다 Read States에 액세스한다.

따라서 Read States 액세스 빈도가 매우 높다. 회사는 Discord의 경험이 매우 경쾌한 것을 목표로 하고 있다.

하지만 Go 구현한 Read States 서비스는 빠른 속도가 필요하다는 요구에 충분히 대응할 수 없었다. 평균으로 하면 고속으로 동작하고 있었지만, 몇 분 간격으로 평균 응답 시간이 갑자기 커져 사용자 경험을 손상했다. 조사를 해보니 이것은 Go의 핵심 기능인 메모리 모델과 가비지 컬렉터(GC)에 기인한 것으로 나타났다.

Go에서 성능이 저하된 이유는 무엇인가?

Go로 기술한 서비스가 성능 목표를 달성 할 수 없었던 이유를 이해하려면 먼저 서비스 데이터 구조와 규모, 액세스 패턴, 아키텍처에 대해 확인할 필요가 있다.

Discord에서 읽기 상태 정보를 저장하는데 사용하는 데이터 구조를 아래에서는 편의적으로 "Read State" 라고 부른다. Discord에는 수십억 Read State가 있다. 사용자와 채널 조합에 대해 하나의 Read State가 필요하기 때문이다. 각 Read State는 여러 카운터를 가진다. 예를 들어, 카운터 중 하나는 사용자가 채널 내에 있는 멘션 수를 나타낸다.

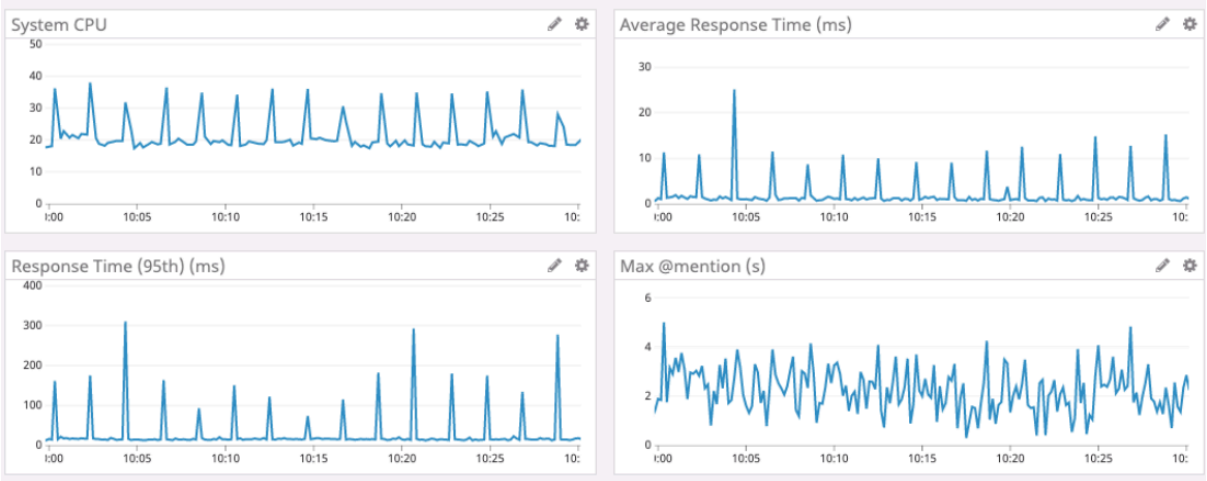
이러한 카운터를 파편화 하게 갱신하면 일관성이 없게된다. atomic하게 갱신해야한다. 또한 빈번하게 0으로 재설정 해야한다.

atomic 카운터를 빠르게 업데이트 하기 위해 각 Read States 서버에는 Read State의 LRU (Least Recently Used) 캐시가 있다. 각 캐시에는 수백만 명의 사용자, 수천만 Read State가 포함된다. 그리고 캐시 업데이트 빈도는 초당 수십만 회에 이른다.

데이터의 지속성을 확보하기 위해 Discord는 분산 데이터베이스 관리 시스템 "Apache Cassandra" 클러스터로 캐시를 보완하고 있다. 캐시 키의 에빅션(불필요한 데이터를

쫓아내는)시 Read State를 데이터베이스에 커밋한다. 또한, Read State를 업데이트 할 때마다 30 초간의 데이터베이스 커밋을 쏟고 있다. 이렇게 데이터베이스에 초당 수만 번의 쓰기가 생긴다.

Go로 구현한 Read States 서비스의 성능을 아래의 그래프에 나타내었다. 최대 멘션 수 (오른쪽)가 불규칙하게 변화하고 있음에도 불구하고 CPU 사용률 (왼쪽)과 평균 응답 시간 (오른쪽)에는 뚜렷한 움직임이 나타나고 있으며, 약 2 분마다 CPU 사용량과 평균 응답 시간이 스파이크로 급증하고 있는 것으로 나타났다.



캐시 키의 에빅션 조치를 실행해도 Go 버전 Read States 서비스는 즉시 메모리 공간을 해제하지 않는다. 대신, 가비지 컬렉터를 자주 실행하여 더 이상 참조가 없어진 메모리 영역을 찾아서 해제한다. 즉, 메모리 공간을 사용하지 않는 직후에 해제하는 것이 아니라 정말 사용되지 않는지를 가비지 컬렉터가 판단 될 때까지 잠시 기다린다. 또한 가비지 컬렉션의 실행 가능한 메모리 영역을 판단하기 위해 다양한 작업을 진행해야 하기 때문에 프로그램의 속도가 저하 될 수 있다.

이러한 이유로 앞의 그래프에 나타났던 평균 응답 시간의 급상승은 분명히 가비지 컬렉션 성능을 위한 것이라고 생각했다.

하지만 Discord의 Go 코드는 매우 효율적으로 기술되어 있었다. 메모리 할당 횟수도 충분히 적게 되어 있다. 즉 가비지 컬렉션에 문제가 있다는 것은 이상하다고 Discord 개발자는 판단했다.

그래서 Go 자체의 소스 코드를 조사한 결과, 적어도 2 분마다 가비지 수집을 강제로 실행하는 것으로 나타났다. 그렇다면 가비지 컬렉터의 동작을 바꾸면 어떻게 될까. 즉시 "GC Percent"의 값을 바꾸어 보았는데, 어떤 값을 설정하더라도 동작은 변하지 않았다. 이유는 Discord 측에 있었다. 가비지 수집이 자주 발생하도록 메모리를 빠르게 할당 할 수 없었기 때문이다.

추가 조사를 계속 한 결과, 2 초마다 거대한 스파이크가 발생하는 다른 이유가 있는 것으로 나타났다. 메모리 공간이 정말 참조 되지 않게 되었는지 여부를 가비지 컬렉터가 판단 할 때, LRU 캐시 전체를 스캔 할 필요가 있었던 것이다.

즉 동작을 빠르게하려면 LRU 캐시를 작게 하면 된다. LRU 캐시의 크기를 변경하기 위한 다른 설정을 서비스에 추가하고, 각 서버에 다수의 분할된 LRU 캐시를 갖도록 구조를 변경했다. 그러자 가베지 컬렉션에 의한 스파이크가 실제로 작아졌다.

하지만이 솔루션에는 부작용이 있다. LCU 캐시의 크기를 작게 하면 사용자의 읽기 상태가 캐시에 포함 될 확률이 낮아져, 데이터베이스 참조를 하기 위해 지연이 생겼다.

Rust을 도입하기 전에는 LCU 캐시 크기와 지연의 균형점을 어떻게든 2초마다 스파이크가 작아 지도록 운용하고 있었다.

Rust 왜 빠른가?

Rust는 가비지 수집이 필요하지 않다**.** 회사가 **Read States** 서비스를 **Rust**에서 구현하려고 생각한 것은 이것이 이유다. Rust를 사용하면 Go에서 구현한 경우에 발생한 평균 응답 시간의 급증은 볼 수 없을 것이다.

Rust는 "메모리 소유권" 이라는 개념을 도입한 비교적 독특한 메모리 관리 방식을 채택하고 있다. 메모리에 소유권이 있기 때문에 어느 메모리 영역에 대한 읽기 및 쓰기 권한을 추적하고 이 메모리 공간이 필요하지 않을 때 즉시 해제한다. Rust는 컴파일 시에 메모리 규칙을 적용하기 때문에 메모리 버그는 거의 발생하지 않는다. C와 같은 수동으로 메모리 영역을 추적 할 필요가 없고, 컴파일러가 모든 걸 관장한다.

이러한 점에서 Read States 서비스의 Rust 버전에서는 LRU 캐시에서 Read State 에빅션 동작이 이루어지면 Read State가 사용하던 메모리 영역은 즉시 해제된다. Rust는 Read State 메모리가 사용되지 않게된 것을 인식하고 즉시 해제하지만, 메모리를 해제할지 여부를 결정하는 런타임 프로세스도 존재하지 않는다.

"비동기 Rust"의 완성도는 낮았지만, 커뮤니티의 협력을 얻었다

Read States 서비스를 다시 구현했을 당시 Rust 안정 버전에서는 비동기 Rust의 지원이 부족했다. 하지만 네트워크 서비스에서는 비동기 프로그래밍이 필수적이다. 비동기 Rust를 활성화한 커뮤니티 버전의 라이브러리가 몇개 존재했지만, 이용 시 복잡한 절차가 필요하고, 문제가 발생한 경우 오류 메시지는 매우 알기 어려웠다.

다행히도, Rust 팀은 비동기 프로그래밍이 용이하도록 적극적으로 노력하고 있으며, Rust의 나이트리 채널에서 비동기 프로그래밍 기능이 강화된 불안정 버전을 사용할 수 있게 되었다.

Discord는 나이트리 출시를 도입하고, 문제가 발생 했을 때 Rust 팀과 협력하여 대처했다. 그 결과, 현재 Rust 안정 버전은 비동기 Rust를 지원하고 있다.

최적화 없이 성능을 발휘한 Rust 버전

Read States 서비스 코드를 다시 작성하는 작업은 매우 원활하게 진행 되었다고 Discord의 개발 팀은 평가하고 있다. 먼저 Go에서 Rust로 코드를 대략적으로 변환하고 슬림이 활성화된 곳에서는 슬림화를 실시했다.

예를 들어, Rust는 우수한 타입 시스템을 갖추고 있어서 어떤 타입이라도 받는 제네릭을 폭넓게 지원하고 있다. 따라서 제네릭을 갖추지 못한 Go에서 필요로 했던 해당하는 코드가 불필요 하게 되었다. 또한 Rust 메모리 모델을 활용함으로써 메모리 보호에 관한 Go 코드의 일부도 불필요하게 되었다.

로드 테스트를 시작했는데, 곧 좋은 결과가 얻어졌다. Rust 버전 Read States 서비스의 평균 응답 시간은 Go 버전 수준이었다. 또한 스파이크가 발생하지 않는다.

주목해야 할 것은, Rust 버전에서는 당초 기본적인 최적화만 수행했다는 것이다. 즉, 기본적인 최적화를 한만큼의 Rust 버전이 수동으로 철저하게 튜닝되어 있던 Go 버전의 성능을 상회하게 된다. 이것은 Rust로 효율적인 프로그램을 쓰는 것이 얼마나 쉬운지 보여주는 좋은 사례이다.

하지만 Discord 개발팀은 이 결과에 만족하지 않고 프로파일링과 성능 최적화를 진행했다. 조금 최적화된 것만으로 Rust 버전은 평균 응답 시간 및 CPU 사용량, 메모리 사용량 등 모든 성능 지표가 Go 버전보다 더 나은 결과를 보여 주었다.

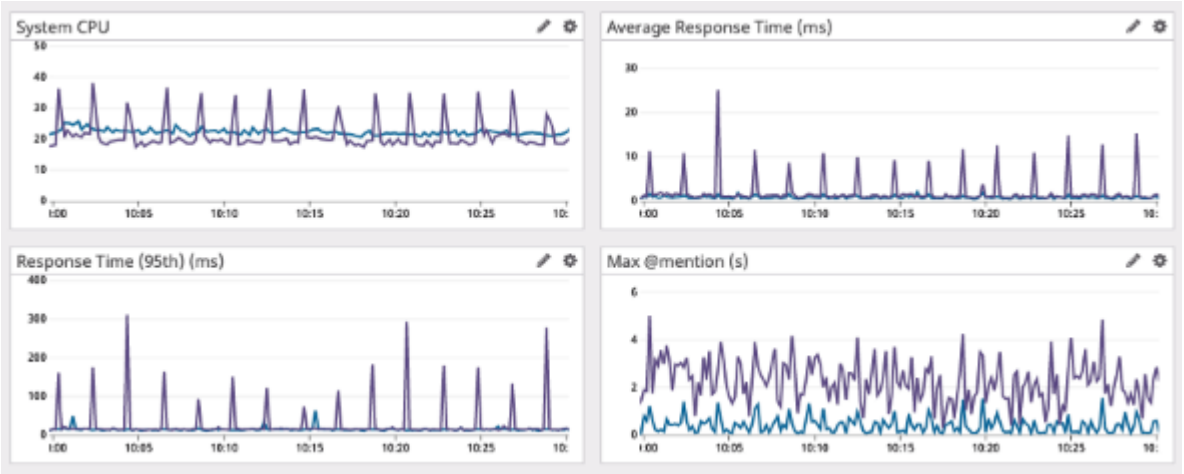
구체적인 최적화의 내용을 나타내면 아래와 같다.

1. LRU 캐시 HashMap를 BTreeMap로 변경하였다(메모리 사용량 최적화를 위한)
2. 초기의 메트릭 라이브러리를 현대적인 Rust의 동시성을 사용하는 것으로 변경했다
3. 메모리 복사 횟수를 줄였다

이러한 최적화를 통해 만족스러운 결과를 얻을 수 있었기 때문에, Discord는 Read States 서비스 Rust 버전 출시를 결정했다.

로드 테스트를 걸쳤기 때문에 릴리스 할 때 문제는 거의 일어나지 않았다. Read States 서비스를 먼저 싱글 카나리아 노드로 확장하고 몇 개의 엣지 케이스가 발견되어서 수정을 실시했다. 얼마 지나지 않아 서비스의 본격 가동을 시작했다.

아래에 Read States 서비스 Go 버전과 Rust 버전에서 CPU 사용률과 평균 응답 시간을 비교한 그래프를 나타낸다. 보라색은 Go 버전, 파랑이 Rust 버전이다. Rust 버전은 CPU 사용률의 변화가 거의없고, 평균 응답 시간에 스파이크가 없는 것으로 나타났다.



캐시 용량을 원래대로 되돌림

Discord 서비스를 대체하고 나서 며칠 후, LRU 캐시 용량을 되돌릴 필요가 있다고 개발팀은 판단했다. Go 버전에서는 앞의 설명처럼, LRU 캐시 크기를 늘리면 가비지 수집에 시간을 요한다. 하지만 Rust 버전에서는 가비지 컬렉션은 필요 없다. 그래서 LCU 캐시 크기 제한을 끌어 올려 성능을 더욱 향상 할 수 있다고 생각했다.

박스 부분의 메모리 용량을 늘려, 더 적은 메모리로 끝나도록 데이터 구조를 최적화 하고, Read State가 800 만개 들어가는 크기로 LCU 캐시의 용량을 늘렸다.

그 결과, CPU 사용률은 10 포인트 정도 감소하고, 평균 응답 시간은 10분의 1 정도까지 떨어졌다.

Rust의 장점은 무엇인가?

Discord는 소프트웨어 스택의 많은 부분에서 Rust을 사용하고있다. 게임 SDK와 Go Live 비디오 캡처 및 인코딩 Elixir NIF, 몇몇 백엔드 서비스 등이다. 새로운 프로젝트와 소프트웨어 구성 요소를 고려할 때, Rust를 사용할 수 있는지 여부를 검토하여 다른 프로그래밍 언어보다 나은 결과를 기대할 수 있는 경우 이용하고 있다고 한다.

Rust를 다양한 프로젝트에 이용해서 알게 된 것은 성능 이외의 장점도 크다는 것이다.

형식 안전성과 차용 검사기(borrow checker)는 특히 도움이 된다. 왜냐하면 제품 요구 사항이 변경 되었을 때 등에 코드 리팩토링 용이하기 때문이다. Rust의 에코 시스템과 도구에도 도움을 받고 있으며, 커뮤니티 활동도 활발하다고 한다.