УДК 004

Авельцов Д.О., ***dmitry.aveltsov@gmail.com***
Институт машиноведения и автоматики НАН КР, Бишкек, Кыргызстан

ИСПОЛЬЗОВАНИЕ WORKER POOL ДЛЯ ОБРАБОТКИ ВХОДЯЩИХ СОЕДИНЕНИЙ В РАСПРЕДЕЛЕННЫХ МИКРОСЕРВИСАХ

Описывается применение подхода с использованием пула воркеров для обработки входящих соединений в микросервисах на языке Go. Рассмотрены структуры Connection Pool Server и Worker Pool Server, их инициализация и обработка соединений. Описаны основные преимущества использования пула воркеров, включая эффективное использование ресурсов, масштабируемость, управление временем ожидания и гибкость в настройке параметров. Также проанализированы различия между Connection Pool Server и Worker Pool Server в контексте производительности, управления нагрузкой, времени ожидания и ресурсных ограничений.

Ключевые слова: микросервисы, Go, Worker Pool, Connection Pool, обработка запросов.

Введение

Сенсорные устройства играют все более важную роль в нашей жизни – от умных домов до промышленных систем мониторинга и управления. С каждым днем количество сенсоров и объем данных, которые они собирают, продолжает расти, что делает эффективную обработку этой информации критически важной задачей.

Для обработки запросов с сенсорных устройств в системе сбора и обработки запросов с сенсорных устройств в разрабатываемых облачных информационно-измерительных системах (ОИИС) мониторинга параметров окружающей среды [1, 2, 3] используется микросервисная архитектура[4]. Сервисы на языке программирования Go разработаны специально для задачи обработки запросов с сенсорных устройств. Однако при таком подходе необходимо эффективно распределять задачи между несколькими потоками, чтобы ускорить обработку запросов и увеличить производительность системы в целом.

В этой статье рассматривается подход к обработке запросов от сенсорных устройств Worker Pool и сравнивается с более простым подходом Connection Pool, оба подхода разработаны для ОИИС. Эти подходы позволяют эффективно использовать ресурсы и распределять задачи между несколькими потоками, что позволяет ускорить обработку запросов и увеличить производительность системы. В статье анализируются плюсы и минусы каждого подхода и представляется код сервиса, который использует оба подхода для обработки данных от сенсорных устройств.

Целью данной статьи является предоставление читателям информации о том, как использование подходов Worker Pool и Connection Pool, специально разработанных для ОИИС, может помочь решить проблемы эффективной обработки запросов от сенсорных устройств. Предоставляются примеры кода, и проводится анализ, как каждый подход влияет на производительность системы.

Горутины и каналы. Для лучшего понимания темы данной статьи следует упомянуть о том, что такое каналы и горутины на языке GO [5].

Горутины – это легковесные потоки, которые реализуют конкурентное программирование в Go. Их называют легковесными потоками, потому что они управляются рантаймом языка, а не операционной системой. Стоимость переключения контекста и расход памяти намного ниже, чем у потоков ОС [6].

Go использует свой собственный планировщик, который способствует горутинам мультиплексироваться по потокам, что позволяет избежать издержек на переключение контекста.

Одним из отличий между горутинами и потоками является то, что у горутин нет доступной программисту идентификации. Вместо этого Go использует абстракцию

локальной памяти потока, которая глобально отображает значения для каждого потока. Это предотвращает злоупотребление локальной памятью потока программистами.

Кроме того, планировщик Go определяет, сколько потоков операционной системы может одновременно выполнять код Go. Это значение по умолчанию равно количеству логических ядер процессора, но может быть настроено в соответствии с потребностями приложения. Таким образом, горутины позволяют создавать эффективные и масштабируемые программы, использующие параллельную обработку [7].

Каналы – это механизм передачи данных между горутинами в языке программирования Go. Они представляют собой абстракцию, которая обеспечивает безопасную и эффективную коммуникацию между горутинами, используя синхронизацию и блокировку.

Каналы могут быть созданы с помощью встроенной функции make() и инициализированы с определенной емкостью буфера. Когда горутина отправляет значение в канал, она блокируется до тех пор, пока другая горутина не получит значение из канала. Если канал имеет емкость буфера, отправка может быть произведена без блокировки до тех пор, пока буфер не заполнен [8].

Каналы также могут использоваться для синхронизации между горутинами. Горутина может ожидать получения сообщения из канала перед продолжением своей работы. Это может быть полезно, например, когда горутина должна завершиться только после того, как другая горутина завершит свою работу.

Каналы в языке Go также могут быть использованы для организации многопоточной обработки данных. Например, можно создать горутину, которая читает данные из канала, обрабатывает их и отправляет результаты в другой канал для дальнейшей обработки. Это позволяет создать эффективную и масштабируемую систему обработки данных. В языке Go каналы являются одним из основных механизмов для организации параллельного выполнения. Они обеспечивают безопасную и эффективную коммуникацию между горутинами и позволяют создавать сложные системы параллельной обработки данных [9].

Основной код сервиса и его анализ. Прежде чем начать рассматривать код, следует отметить, что он был сокращен и модифицирован для облегчения понимания и что некоторые зависимости были опущены для сохранения краткости.

Код сервиса можно разделить на несколько основных блоков:

1. Обработчики TCP и HTTP запросов.
2. Работа с RabbitMQ [10] и обмен сообщениями.
3. Использование Worker Pool Server для обработки подключений.
4. Общая структура сервиса и инициализация всех компонентов.

Основное внимание будет сосредоточено на пункте 3, так как именно здесь реализована основная логика работы с воркерами.

Работа с worker'ами и сравнение альтернатив. Основные альтернативы выбранного подхода с использованием worker pool включают:

1. Обработка запросов последовательно без использования worker pool.
2. Использование горутин для каждого входящего соединения без ограничений на их количество.
3. Connection pool: использование горутин для каждого входящего соединения с ограничением на их количество при помощи буферизации соединений.

Обработка запросов последовательно. При последовательной обработке запросов сервис будет принимать и обрабатывать каждый запрос по очереди. Этот подход прост в реализации, однако он не эффективен при большой нагрузке, поскольку запросы будут обрабатываться медленно и могут вызвать задержки.

Использование горутин для каждого входящего соединения. Использование горутин для каждого входящего соединения позволяет обрабатывать несколько запросов одновременно, увеличивая производительность. Однако при большой нагрузке это может привести к чрезмерному потреблению ресурсов и деградации производительности.

Проведем более детальное сравнение connection pool и выбранного подхода worker pool.

Connection pool

Основная идея этого подхода – создать фиксированное количество обработчиков, которые обрабатывают входящие соединения за счет буферизации соединений. Для каждого соединения создается обработчик. Если буфер соединений заполнен, новое соединение ожидает своей очереди.

Преимущества:

- Простота реализации.

Недостатки:

- Может приводить к накладным расходам на создание и удаление большого количества горутин, особенно при большом объеме входящего трафика.

Worker Pool

Данный подход предполагает динамическое создание и уничтожение обработчиков на основе загруженности сервера и заданного количества обработчиков. Если все обработчики заняты и количество воркеров не превышает максимально допустимое, создается новый обработчик. В противном случае соединение ожидает своей очереди.

Преимущества:

- Гибкость: автоматическая адаптация к изменению нагрузки на сервер.

Недостатки:

- Усложнение логики сервера.

Анализ кода Connection pool

Листинг Connection Pool Server:

```
type ConnectionPoolServer struct {
  logger   *zerolog.Logger
  connPool chan struct{}
}
func NewConnectionPoolServer(poolSize int) *ConnectionPoolServer {
  logger := logging.GetLogger()
  return &ConnectionPoolServer{connPool: make(chan struct{}, poolSize), logger: &logger}
}
func (s *ConnectionPoolServer) Serve(listener net.Listener, handler ConnectionHandler) {
  for {
    conn, err := listener.Accept()
    if err != nil {
      if err != cmux.ErrListenerClosed {
        s.logger.Panic().Err(err).Msg("Error while accepting connection")
      }
      s.logger.Error().Err(err).Msg("Listener closed")
      return
    }
    s.connPool <- struct{}{}
    go func() {
      defer func() {
        <-s.connPool
      }()
      handler.HandleConnection(conn)
```

```
    }()
  }
}
```

Структура Connection Pool Server (рис. 1)

Сервер определяется структурой Connection Pool Server, содержащей следующие поля:

- logger: объект логгера из библиотеки zerolog для записи сообщений о событиях сервера.
- Conn Pool: канал, представляющий пул соединений. Его размер определяет максимальное количество одновременно обрабатываемых соединений.

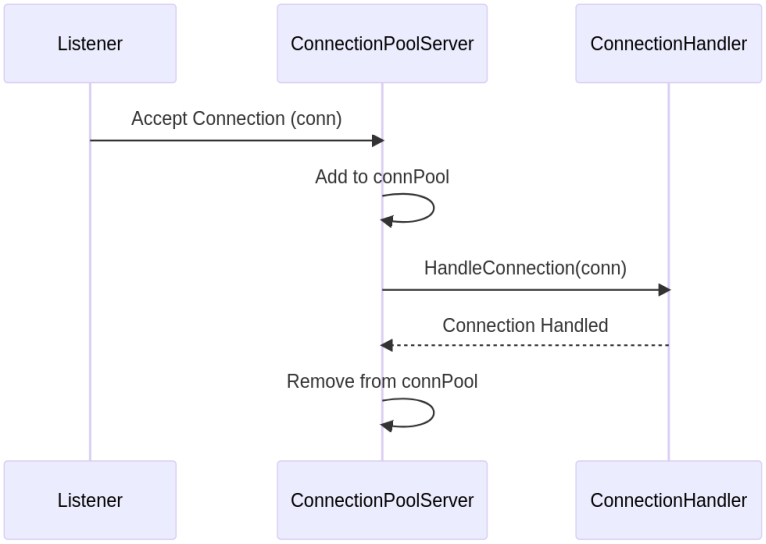

Рисунок 1 – Диаграмма последовательности процесса работы сервера

Функция New Connection Pool Server

Функция New Connection Pool Server создает новый экземпляр сервера с заданным размером пула соединений. Входной параметр pool Size определяет максимальное количество одновременно обрабатываемых соединений.

Метод Serve

Метод Serve слушает входящие соединения и обрабатывает их с использованием пула соединений. При получении нового соединения оно отправляется в пул, и, если в пуле имеется свободное место, запускается новая горутина для обработки соединения с помощью переданного обработчика handler.

Если возникает ошибка при принятии соединения и ошибка не связана с закрытием слушателя, сервер записывает сообщение об ошибке в лог и завершает свою работу. В случае закрытия слушателя сервер также записывает сообщение в лог и завершает работу.

Обработка соединений (рис. 2).

Для каждого принятого соединения создается новая горутина, которая обрабатывает соединение с помощью переданного обработчика handler. После завершения обработки соединения горутина освобождает место в пуле соединений, что позволяет принимать новые соединения.

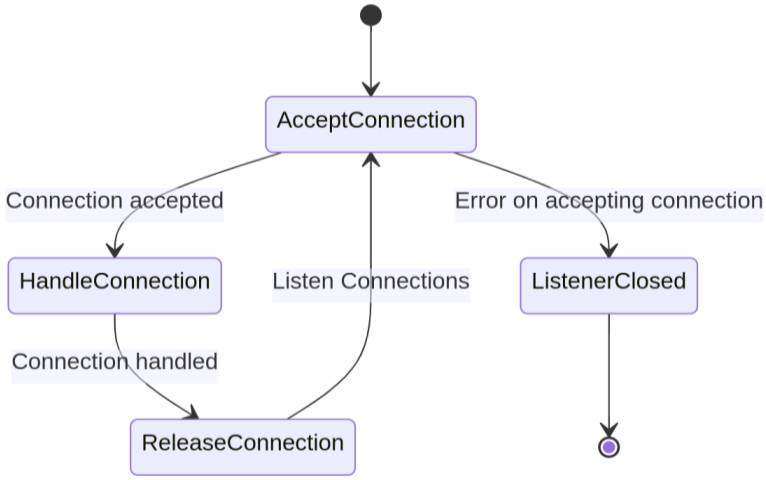

Рисунок 2 – Алгоритм процесса обработки соединений

В целом рассмотренный код представляет собой простую и эффективную реализацию сервера с ограниченным буфером соединений на языке Go. Он обеспечивает масштабируемость, стабильность и отказоустойчивость при обработке входящих соединений.

Анализ кода worker pool

Worker представляет собой поток, который обрабатывает запросы. Для работы с воркерами используется пул воркеров. Каждый воркер является отдельным процессом, который обрабатывает запросы в отдельном потоке. В зависимости от количества запросов, которые нужно обработать, количество воркеров может изменяться динамически (рис. 3).

Для организации работы с воркерами в проекте используется пакет "internal". Он содержит в себе два подпакета – "connection" и "server". Пакет "connection" отвечает за обработку подключений к сервису, а пакет "server" – за создание пула воркеров и организацию работы с ними.

Листинг Worker Pool Server:

```
type WorkerPoolServer struct {
  workersPoolSize     int32
  initialWorkers      int32
  connectionPoolSize  int
  idleTimeoutDuration time.Duration
  logger              *zerolog.Logger
}
func NewWorkerServer(workers, initialWorkers int32, connectionPoolSize int, idleTimeoutDuration time.Duration) *WorkerPoolServer {
  if workers < initialWorkers {
    panic("workersPoolSize must be greater or equal to initialWorkers")
  }
  logger := logging.GetLogger()
  return &WorkerPoolServer{
    workersPoolSize:     workers,
    initialWorkers:      initialWorkers,
    connectionPoolSize:  connectionPoolSize,
    idleTimeoutDuration: idleTimeoutDuration,
    logger:              &logger,
  }
```

```
}
func (s *WorkerPoolServer) Serve(listener net.Listener, handler ConnectionHandler) {
  connectionCh := make(chan net.Conn, s.connectionPoolSize)
  var workerCounter int32
  for i := int32(0); i < s.initialWorkers; i++ {
    atomic.AddInt32(&workerCounter, 1)
    go s.worker(connectionCh, handler, &workerCounter)
  }
  for {
    // if blocked check SOMAXCONN value in /proc/sys/net/core/somaxconn
    conn, err := listener.Accept()
    if err != nil {
      if err != cmux.ErrListenerClosed {
        s.logger.Panic().Err(err).Msg("Error while accepting connection")
      }
      s.logger.Error().Err(err).Msg("Listener closed")
      return
    }
    select {
    case connectionCh <- conn:
      // pass the connection to the available worker
    default:
      if atomic.LoadInt32(&workerCounter) < s.workersPoolSize {
        atomic.AddInt32(&workerCounter, 1)
        go s.worker(connectionCh, handler, &workerCounter)
      } else {
        connectionCh <- conn
      }
    }
  }
}

func (s *WorkerPoolServer) worker(connectionCh chan net.Conn, handler
ConnectionHandler, workerCounter *int32) {
  defer atomic.AddInt32(workerCounter, -1)
  timeout := time.NewTimer(s.idleTimeoutDuration)
  for {
    select {
    case conn, ok := <-connectionCh:
      if !ok {
        return
      }
      if !timeout.Stop() {
        select {
        case <-timeout.C:
        default:
        }
      }
      handler.HandleConnection(conn)
      timeout.Reset(s.idleTimeoutDuration)
    case <-timeout.C:
      if atomic.LoadInt32(workerCounter) > s.initialWorkers {
```

```
            // worker timeout, closing worker
            return
        }
    }
  }
}
```

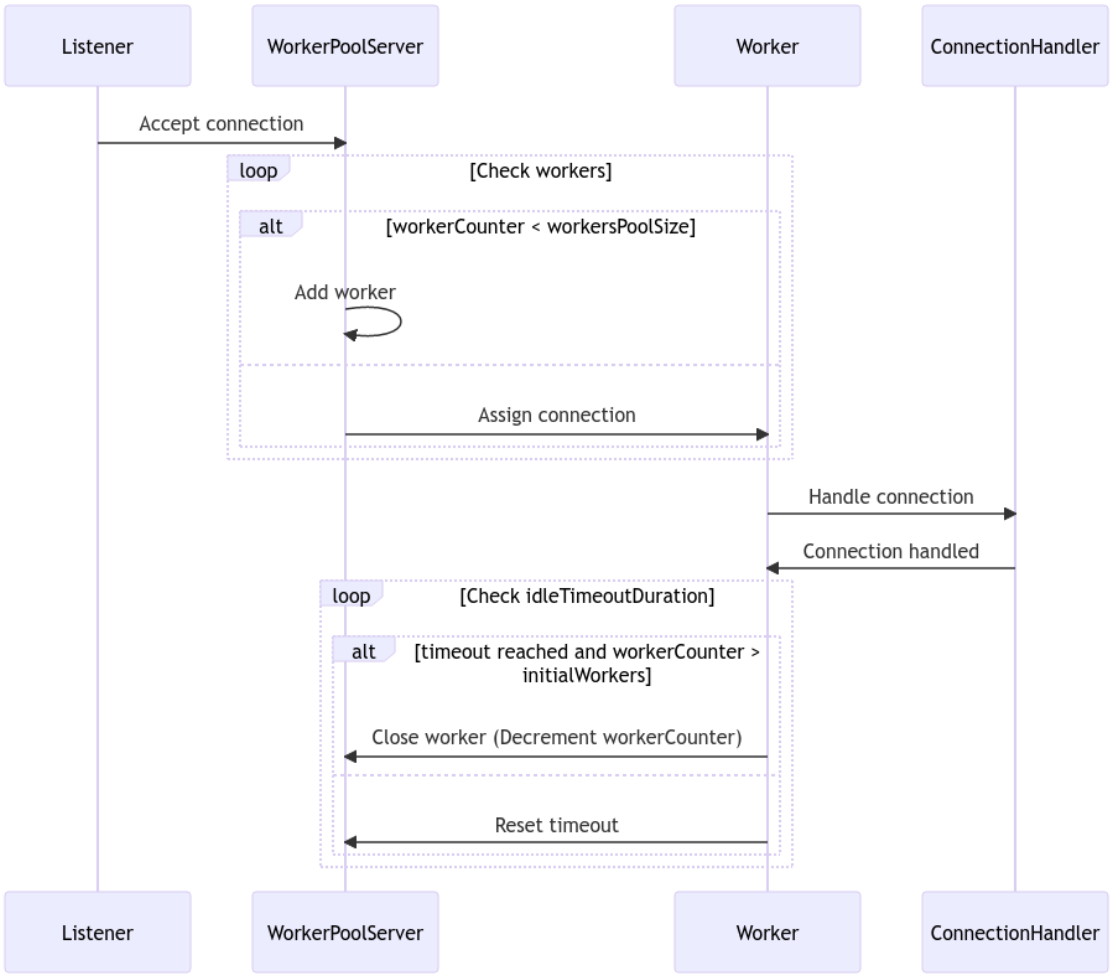

Рисунок 3 – Общая последовательность работы Worker Pool Server

1. Обзор структуры Worker Pool Server

Структура Worker Pool Server содержит следующие поля:

- workers Pool Size: максимальный размер пула воркеров.
- Initial Workers: начальное количество воркеров, создаваемых при запуске микросервиса.
- Connection PoolSize: размер пула соединений (очередь соединений, ожидающих обработки воркерами).
- Idle Timeout Duration: время ожидания воркера без активности, после которого он будет завершен.
- logger: объект для логирования.

2. Инициализация Worker Pool Server (рис. 4)

Функция New Worker Server создает и возвращает новый экземпляр структуры Worker Pool Server с указанными параметрами. Важно заметить, что функция проверяет, чтобы значение workers Pool Size было больше или равно initial Workers. В противном случае функция вызовет панику.

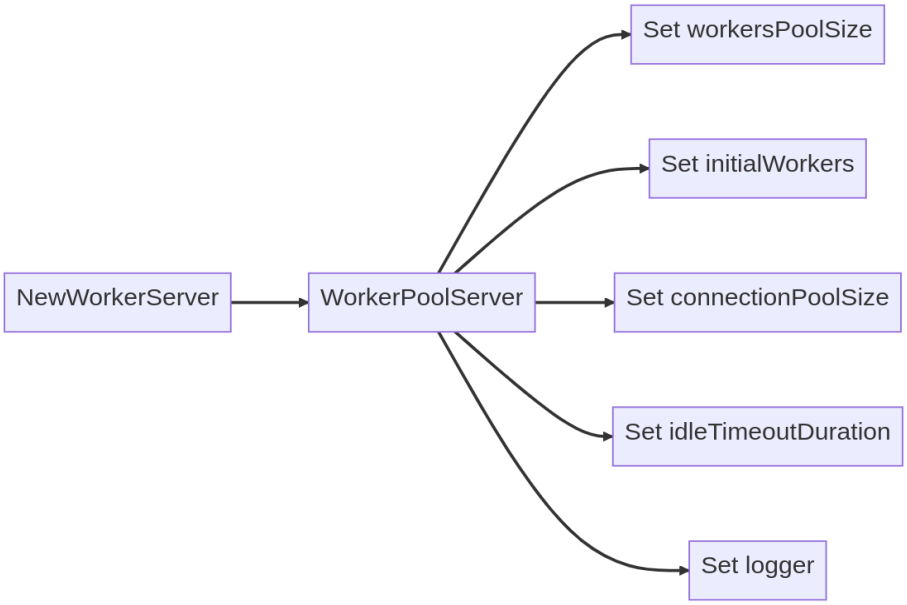

Рисунок 4 – Инициализация Worker Pool Server

3. Запуск микросервиса

Метод Serve запускает микросервис, прослушивает входящие соединения и передает их воркерам через канал connection Ch. Метод создает и запускает начальное количество воркеров, указанное в поле initial Workers.

Когда новое соединение поступает, метод пытается отправить его в канал connection Ch. Если канал заполнен, метод проверяет, не превышено ли значение workers Pool Size. Если нет, создается новый воркер. В противном случае соединение добавляется в канал connection Ch, чтобы быть обработанным следующим свободным воркером (рис. 5).

4. Обработка соединений воркерами (рис. 6).

Функция worker представляет собой горутину, которая обрабатывает соединения, полученные из канала connection Ch. Воркер получает соединение и вызывает функцию handler.Handle Connection(conn) для обработки соединения. Если воркер простаивает дольше, чем idle Timeout Duration, он завершается, освобождая системные ресурсы.

Реализация демонстрирует подходы к управлению пулом воркеров, обеспечивая оптимальное использование системных ресурсов и быстрое обслуживание клиентских запросов.

Основные преимущества использования пула воркеров в данном коде включают следующее:

1. Эффективное использование ресурсов: пул воркеров позволяет микросервису использовать доступные ресурсы процессора и памяти более эффективно, тем самым обеспечивая лучшую производительность.

2. Масштабируемость: воркеры могут быть динамически добавлены или удалены в зависимости от текущей нагрузки, что делает микросервис более масштабируемым и устойчивым к непредвиденным всплескам нагрузки.

3. Управление временем ожидания: время ожидания воркеров позволяет автоматически освобождать системные ресурсы при пониженной нагрузке, улучшая эффективность использования ресурсов.

4. Легковесность и гибкость: реализация на языке Go обеспечивает легковесность и высокую производительность микросервиса, а также гибкость в настройке параметров, таких как размер пула воркеров и времени ожидания.

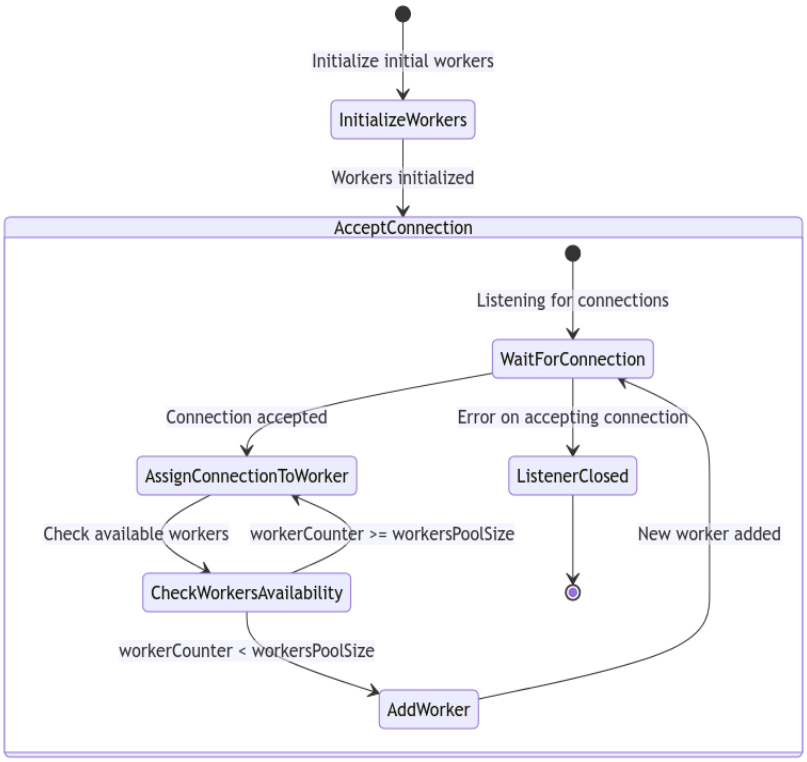

Рисунок 5 – Алгоритм работы сервера с пулом воркеров

Для дополнительной оптимизации и настройки данной реализации микросервиса можно провести нагрузочное тестирование, мониторинг и анализ производительности системы, чтобы определить оптимальные значения параметров конфигурации. Это позволит микросервису адаптироваться к изменяющимся условиям и обеспечить наилучшую производительность и устойчивость системы.

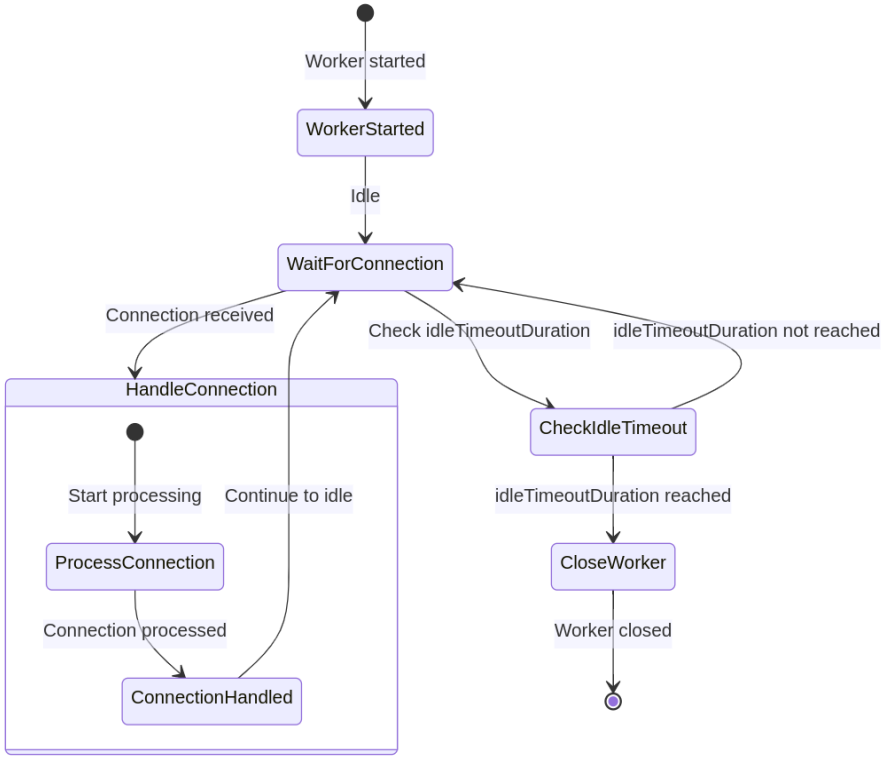

Рисунок 6 – Диаграмма состояний воркеров с учетом таймаута ожидания

Сравнение Worker Pool Server и Connection Pool Server

Производительность. Подход с использованием Worker Pool Server может обрабатывать соединения параллельно и масштабировать число воркеров в зависимости от нагрузки. Это приводит к улучшению производительности по сравнению с подходом, использующим Connection Pool Server, где обработка запросов выполняется параллельно, но без контроля над горутинами и их временем работы.

Управление нагрузкой. Worker Pool Server может лучше справляться с варьирующейся нагрузкой благодаря динамическому изменению количества воркеров и использованию каналов для передачи соединений. Connection Pool Server не имеет явных средств управления нагрузкой, что может привести к проблемам при пиковых нагрузках.

Таймауты и ресурсные ограничения. В подходе Worker Pool Server воркеры имеют таймауты ожидания, что позволяет им освободить ресурсы и завершить работу, если соединения не поступают. Это обеспечивает лучший контроль над ресурсами по сравнению с Connection Pool Server, где нет явного управления временем ожидания и ресурсами.

Сложность. Connection Pool Server является более простым с точки зрения кода и архитектуры, поскольку он не использует пул воркеров и каналы для управления соединениями. Однако эта простота может привести к проблемам с производительностью и управлением нагрузкой.

Заключение

В данной статье было рассмотрено применение пула воркеров для обработки входящих соединений в микросервисах на языке Go. Было выявлено, что использование пула воркеров обеспечивает более эффективное использование ресурсов, масштабируемость, управление временем ожидания и гибкость в настройке параметров. Однако также были проанализированы различия между Connection Pool Server и Worker Pool Server и было обнаружено, что каждый подход имеет свои особенности и может быть более или менее подходящим в зависимости от конкретных требований приложения. Таким образом, выбор между Connection Pool Server и Worker Pool Server должен быть основан на конкретных потребностях микросервиса и его условиях эксплуатации. В целом применение пула воркеров представляет собой эффективное и гибкое решение для обработки входящих соединений в микросервисах на языке Go, которое может обеспечить высокую производительность и устойчивость системы.

Литература

1. Авельцов, Д. О. Разработка модуля виртуализации сенсорных устройств для распределенных информационно-измерительных систем / Д. О. Авельцов, В. В. Гайдамако // Проблемы автоматики и управления. – 2020. – № 1(38). – С. 89 – 103. – DOI 10.5281/zenodo.3904148. – EDN IDWEPH.

2. Гайдамако В.В. Разработка прототипа модели облачных информационно-измерительных систем с использованием библиотеки SIMGRID //Математическое и компьютерное моделирование: сборник материалов IX Международной научной конференции, посвященной 85-летию проф. В.И. Потапова (20 ноября 2021г.). – Омск: Изд-во Омск. гос. ун-т, 2021. – С.233 – 235.

3. Разработка Web-портала экологической информации Кыргызской Республики / В. В. Гайдамако, Б. К. Каныбеков, Н. М. Лыченко, Д. А. Текеев // Проблемы автоматики и управления. – 2022. – № 3(45). – С. 74 – 83. – EDN QYSVOQ.

4. Микросервисная архитектура. https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81%D0%BD%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0 (дата обращения 25.02.2023)

5. Язык Go. https://ru.wikipedia.org/wiki/Go (дата обращения 25.02.2023)
6. Горутины –основы Go. https://ru.hexlet.io/courses/go-basics/lessons/goroutines/theory_unit (дата обращения 25.02.2023)
7. Отличия горутин от потоков. https://backendinterview.ru/goLang/concurrency/gouritine (дата обращения 25.02.2023)
8. What are buffered channels in Golang? Vafa Batool. https://www.educative.io/answers/what-are-buffered-channels-in-golang (дата обращения 25.02.2023)
9. An Introduction to Channels in Go. https://www.sohamkamani.com/golang/channels (дата обращения 25.02.2023)
10. RabbitMQ. https://ru.wikipedia.org/wiki/RabbitMQ (дата обращения 25.02.2023).