University of Tartu

Faculty of Mathematics and computer science

Report of distributed Systems Seminar

TweedlyNews

Author: Taivo Käsper

Supervised by: Oleg Batrašev

Tartu 2014

Table of Contents

Introduction

Over time parallel programming has become more and more important because processors can not go any much faster but need to start doing work in parallel. To harness the multi core processors, programs have to be written in a way that support parallel computing using multiple threads. For a long time there has been no revolution on making multi threaded programming more simpler until now when functional programming has become very popular and there are communities which enforce that code should be simple and more similar to natural languages. This is something that has been the main goal of GO language.

This paper will focus on two main GO language paradigms Goroutines and Channels which are specifically created to make concurrent programming simpler. The author tries them out in a real world situation and gives notes on usages and program structure using goroutines.

Goroutines

For concurrent programming GO has implemented goroutines which can be thought of as lightweight threads. Basic idea of goroutine is that multiple instances of them are ran on the same operating system level threads but only one at a time can work, others wait. When the running goroutine blocks then another starts to work. By default GO programming language uses one OS thread but this can be changed with "GOMAXPROCS" parameter. Programmer has to be aware of the algorithms used because if the program itself is not very well parallelisable it gets slower instead because context switches on the OS level threads are expensive [1]. Currently the GO's goroutine scheduler is not as good as it needs to be and is said to become smarter by optimising such cases. At this point it is important to remember that concurrency and parallelism are not the same thing [1].

Goroutines are meant to make concurrency easy to use. The idea, which has been around for a while, is to multiplex independently executing functions onto a set of threads [2]. When a function blocks the run-time automatically moves other function calls on the same operating system thread to a different, runnable thread so they won't be blocked. The programmer sees none of this. The result, which we call goroutines, can be very cheap, just a few kilobytes [2]. To make the stacks small, Go's run-time uses resizable, bounded stacks [2]. When the amount of memory is not enough then the run-time grows or shrinks allocated memory size automatically. The CPU overhead averages about three cheap instructions per function call [2].

Creators of GO programming language encourage to create hundreds of thousands of goroutines. It is not only useful on the performance but it also decreases coupling and encourages modular design.

Buffered and unbuffered Channels

Goroutines themselves do not contain a way for communication, this can easily be fixed by using channels to enable easy data exchange. Channels do not only enable to send data from A to B but also include automatic synchronisation. There are multiple ways to use channel synchronization. Firstly you can use buffered channel which allows a program to write to the channel multiple objects - blocking is done on read from the channel meaning that if there is nothing in the channel then the reading goroutine blocks until someone writes there. This is like an automatic synchronization. Secondly you can use an unbuffered channel where in the channel can only be 1 object at a time.

When choosing whether to use buffered of unbuffered channel programmer has to choose if to increase the throughput or decrease memory consumption. It is important to note that throughput cannot be increased to infinity because if the writing goroutine performs faster than the reading, then the buffer of the channel will fill up. Therefor it is usually wise to use channel with a small buffer size.

Channels in GO are a first-class value that can be allocated and passed around like any other object [3]. This allows to create interesting structures for implementing safe, parallel demultiplexing.

Introduction to TweedlyNews

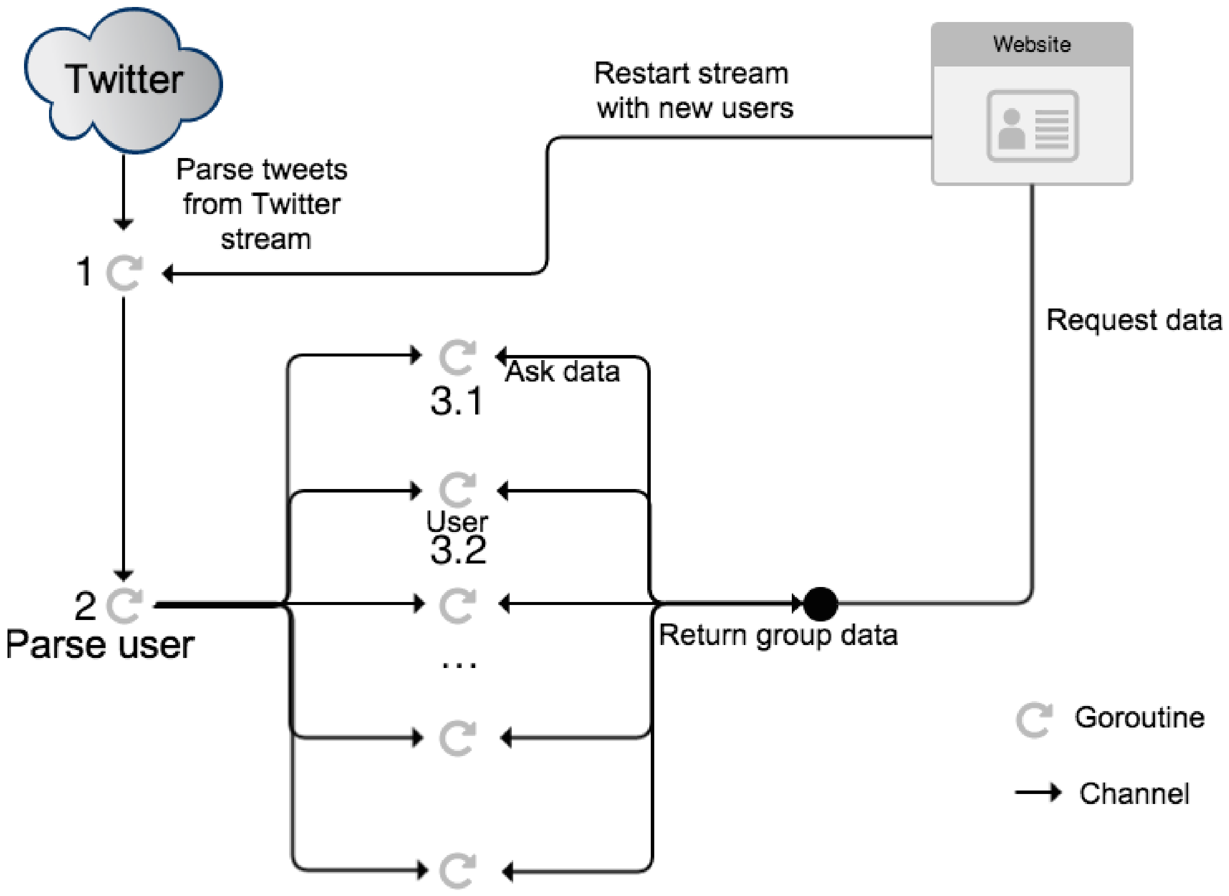
For gaining more experience with GO and getting real experience with goroutines an example application called TweedlyNews was created. The main goal of the program is to enable everybody to created and edit groups of Twitter users. TweedlyNews streams all Tweets from every user from every group and enables to view them in an organised manner. It depends on a community that it administers the groups and provides an opportunity to read interesting news from cool tweeters without the need to knowing who to follow on Twitter. A good example would be that you can create a group called "Tech news" and add only extreme techies there who tweet about new technologies and gadgets the minute they come out.

The application uses GO's built in http web server to handle requests and holds an open http stream with Twitter to get Tweets when they are posted. At first each user had their own goroutine which polled their Twitter wall to find out if anything new is posted but this solution didn't prove to be successful because Twitter blocks applications that poll too much. Therefor Twitter's streaming api was used and a GO library called TwitterApi partially rewritten to enable an active stream with all the users to track. Because Twitter allows only one active stream per application which contains all the selected users, TwitterApi library had to be modified to enable stream restart with new users if a new tweeter was added by the TweedlyNews community.

Source code is available from this BitBucket repository: http://tiny.cc/c6wydx.

TweedlyNews structure

The following image is an illustration of how channels and goroutines are used in TweedlyNews.



Structure of goroutines and channels in TweedlyNews

A stream reading from Twitter is done with goroutine 1 which parses Tweets from the body of the active response from Twitter. This parsing goroutine hands a raw Tweet to another parser goroutine 2 which finds from which tracked user it originates from and then hands it on to a user specific goroutine. Each user has their own goroutine which is responsible for cacheing tweets and serving them if requested.

Goroutine 1 is writing Tweets to channel for goroutine 2 and also has a channel which is only used to notify that a restart with new users is needed. During the restart other goroutines block because there is nothing to read from their input channels. Goroutine 2 finds a corresponding user's channel for adding Tweet and writes there which triggers an add operation. Goroutines 3.1, 3.2, …, 3.n use 3 channels - one for adding, one which is used to notify that data is requested and one for returning data. Web request handler finds the notification channels of needed users and sends a notification there saying that write your data to your output channel. When it has all needed results it writes them to http response. This process is fast because each user’s goroutine returns it’s current state without doing anything else.

Conclusion

Concurrent programming in GO is very enjoyable because their main goal was to make in easier. There definitely are things to improve like running goroutines in parallel and tooling but the overall experience was very pleasing.

Acknowledgments

The author would like to express his thanks of gratitude to Oleg Batrašev, Märt Bakhoff and the Distributed Systems Seminar group.

Citations

1. Why does using GOMAXPROCS > 1 sometimes make my program slower? Retrieved from http://golang.org/doc/faq#Why_GOMAXPROCS
2. Why goroutines instead of threads? Retrieved from http://golang.org/doc/faq#goroutines
3. Channels of channels. Retrieved from http://golang.org/doc/effective_go.html#chan_of_chan