Limiting the Number of Requests

Multiple clients may make their requests to a server all at once, which can result in a massive number of requests that burden the server. This lesson brings the solution to this problem: a workaround in Go.

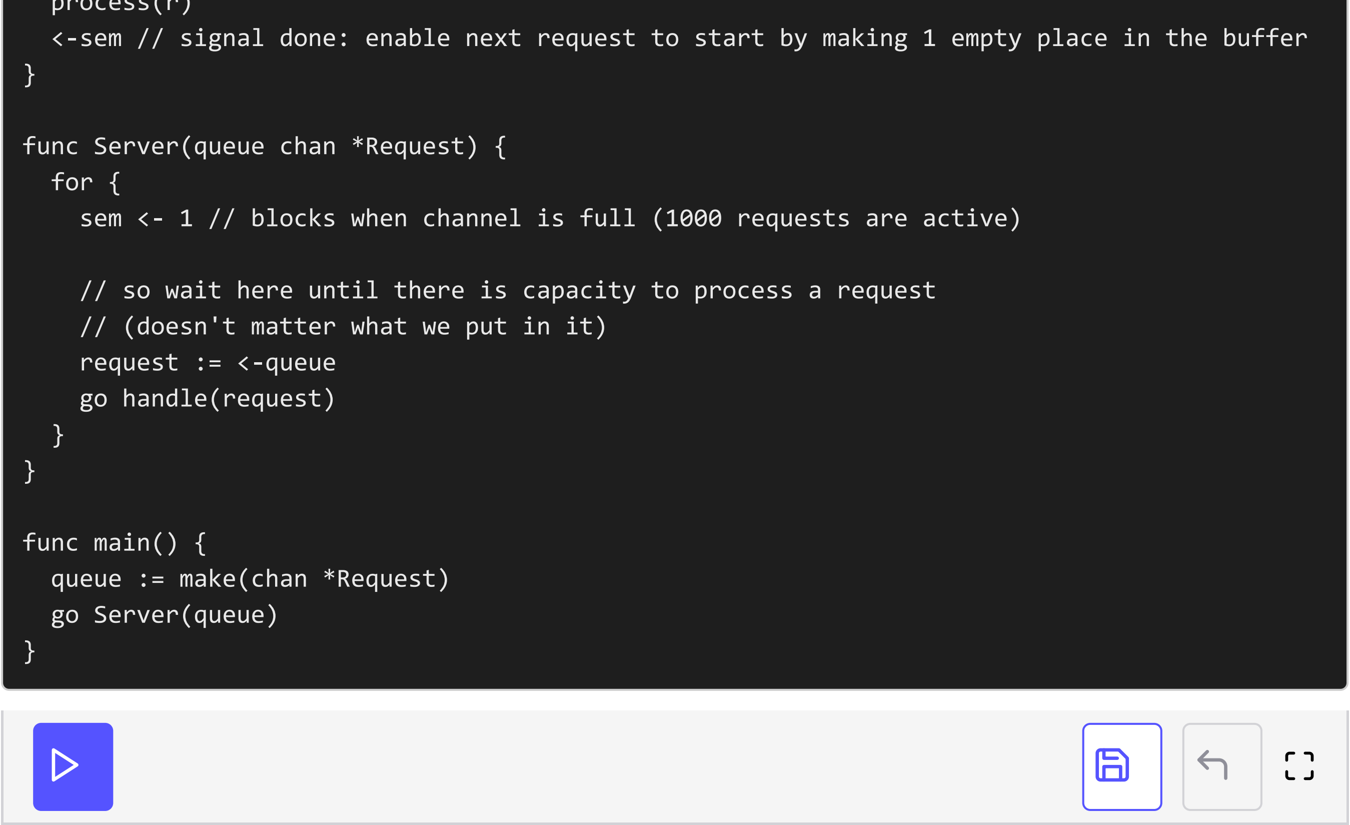
WE'LL COVER THE FOLLOWING

- Bounding requests processed concurrently

Bounding requests processed concurrently

This is easily accomplished using a channel with a buffer, whose capacity is the maximum number of concurrent requests. The following program does nothing useful, but contains the technique to bound the requests. No more than `MAXREQS` requests will be handled and processed simultaneously because, when the buffer of the channel `sem` is full, the function handle blocks and no other requests can start until a request is removed from `sem`. The `sem` acts like a *semaphore* that is a technical term for a flag variable in a program that signals a certain condition.

```go
package main

const (
  AvailableMemory = 10 << 20 // 10 MB, for example
  AverageMemoryPerRequest = 10 << 10 // 10 KB
  MAXREQS = AvailableMemory / AverageMemoryPerRequest // here amounts to 1000
)

var sem = make(chan int, MAXREQS)

type Request struct {
  a, b int
  replyc chan int
}

func process(r *Request) {
  // Do something
  // May take a long time and use a lot of memory or CPU
}

func handle(r *Request) {
  process(r)
```



```
  process(r)
  <-sem // signal done: enable next request to start by making 1 empty place in the buffer
}

func Server(queue chan *Request) {
  for {
    sem <- 1 // blocks when channel is full (1000 requests are active)

    // so wait here until there is capacity to process a request
    // (doesn't matter what we put in it)
    request := <-queue
    go handle(request)
  }
}

func main() {
  queue := make(chan *Request)
  go Server(queue)
}
```

Maximum Tasks

In the constants section from **line 3** to **line 7**, we calculate the *maximum* number of requests `MAXREQS` as the available memory is divided by the memory every request needs. Then, at **line 9**, we make a channel of integers called `sem`, which has a buffer just to that amount. We define a typical `Request` struct (from **line 11** to **line 14**). We define stub functions `process()` and `handle()`, which take a request, process it, and remove an item for `sem` to make a place for the next request (**line 23**).

The `main()` function makes a `queue` of `Request` and starts `Server` with it. The `Server()` function (defined from **line 26** to **line 35**), starts an infinite for-loop, putting **1** on the semaphore channel. Then, at **line 32** and **line 33**, the `Server()` tries to get a request from the `queue`. As soon as it gets one, it starts handling this request in a separate goroutine (**line 33**).

When `sem` is full, **line 28** blocks until a process in action gets a value from `sem` at **line 23**. That way, only a maximum of `MAXREQS` processes can be handled at a time.

Reducing the limit of the number of requests allowed enables the server to provide services in the best manner. Parallelizing the computation over a large amount of data within the limit, optimizes the process. See the next lesson to learn how this works.