The Leaky Bucket Algorithm

This lesson gives an implementation and explanation of an algorithm that controls client-server communication using a buffer.

WE'LL COVER THE FOLLOWING

Introduction

Consider the following *client-server* configuration. The `client` goroutine performs an infinite loop that is receiving data from some source, perhaps a network; the data are read in buffers of type `Buffer`. To avoid allocating and freeing buffers too much, it keeps a free list of them, and uses a buffered channel to represent them:

```
var freeList = make(chan *Buffer, 100)
```

This queue of reusable buffers is shared with the server.

When receiving data, the client tries to take a buffer from `freeList`. If this channel is empty, a new buffer gets allocated. Once the message buffer is loaded, it is sent to the server on `serverChan`:

```
var serverChan = make(chan *Buffer)
```

Example

Here is the algorithm for the client code:

```
func client() {
  for {
    var b *Buffer
```

```
    // Grab a buffer if available; allocate if not
    select {
      case b = <-freeList:
        // Got one; nothing more to do
      default:
        // None free, so allocate a new one
        b = new(Buffer)
    }
    loadInto(b) // Read next message from the network
    serverChan <- b // Send to server
  }
}
```

The server loop receives each message from the client, processes it, and tries to return the buffer to the shared free list of buffers:

```
func server() {
  for {
    b := <-serverChan // Wait for work.
    process(b)
    // Reuse buffer if there's room.
    select {
      case freeList <- b:
      // Reuse buffer if free slot on freeList; nothing more to do
      default:
      // Free list full, just carry on: the buffer is 'dropped'.
    }
  }
}
```

However, this doesn't work when `freeList` is full, in which case the buffer is *dropped on the floor* (hence the name *leaky bucket*) to be reclaimed by the garbage collector.

Before summing up the goroutines and channels, it's important to learn how to benchmark goroutines, provide concurrent access to the objects through channels, and listen through different channels. See the next lesson to get a grasp of these concepts.