Multingenium: Multithreaded, Distributed Computing in Go

Joseph J. Perme
Department of Computer Science
University of Dayton
300 College Park
Dayton, Ohio 45469
jp@jper.me

ABSTRACT

Distributed computing is becoming increasingly important in an ever-connected world. By means of our ever-available interconnectedness known as the Internet, the computing power of many machines can be leveraged for problem solving. This project enhances a multithreaded echo server in Go to enable clients to send messages to the server, which, upon receipt, the server broadcasts to all connected clients, save for the original sender. An interpreter for a language called *Multingenium* (an example of a portmanteau word created from the Latin for 'many brains') was implemented so that clients could send *Multingenium* commands to the server, which then leveraged the computing power of the clients to execute the command.

1. INTRODUCTION

Though this project is on the surface a proof-of-concept, it demonstrates that the framework established can be built upon for more complex work that would be able to take advantage of such a distributed computing model. Cloud (i.e., distributed) computing is an ever-growing, constantly expanding industry, with big-name players such as Amazon and Microsoft pouring millions into the research and development of more efficient, easier-to-use, and faster cloud computing solutions to push its adoption even further. One research group from UC Riverside proposed using cell phones for distributed computing while they are charging overnight, as their computing power typically sits latent for many hours every night [1]. Companies are always seeking to lower costs and raise profits, and outsourcing computational resources is often an area that accomplishes both of these goals. Rather than pouring revenue into independent development of computational data centers, companies are more willing to rent time on specialized cloud computing structures, paying by CPU clock cycle or total time run. As an introduction to distributed computing, this project extends the client-server structure provided by Jan's code to demonstrate leveraging a simple multi-client network for simple work, as a proof-of-concept for a distributed computing pipeline.

CPS 499/592 Emerging Languages, University of Dayton, Spring 2016
© 2016 Copyright held by the owner/author(s).

1.1 A Roadmap of the How

Beginning with the functional, multithreaded echo server from the chapter titled 'Socket-level Programming' from Jan Newmarch's e-book titled 'Network Programming with Go' [3], the first step in the process of extending the provided multithreaded model to support distributed computing is to support continuous communication among clients through the server. The simplest way to demonstrate the support of this communication is by implementing chat functionality. From there, interpretation of broadcast messages (originally simply chat messages) is added to enable users to leverage the computing power of the network through a command set. Unfortunately Go does not —to the best of my knowledge—have an included interpreter. This leaves a couple of options for implementing control of the computing network. A second already-existing language could be leveraged (admittedly an afterthought to the implementation of the following solution,) or an interpreter for a new language could be written, and the network's computational power would then rely on the capabilities of this language and their extension. The latter approach was taken by way of a simple 'language' dubbed *Multingenium* (portmanteau of the Latin for 'many brains'). A client can then send a command to the server with what will be called a 'work set' (the data to manipulate/the environment to operate under,) and the server will handle the interpretation and distribution of the work to be done – the 'command.' Keeping this overview in-mind, the following is a deeper look at the implementation of *Multingenium*.

2. SINGLE-CLIENT TO MULTI-CLIENT

Single-client applications have far less purpose in the online communities than multi-client applications, which is one cause for making this application multithreaded. With a single-threaded application, any server can only respond to a single client at a time, causing the other clients to either queue up, or to simply disconnect and return an error to their operator. This is where the beauty of Go comes into play. Go makes spawning *goroutines* (i.e., lightweight threads) extremely simple. Preface the name of a function with the keyword `go` to start it in its own goroutine. For example, in `server.go`, the following is used to spawn a thread for each new client that attempts to connect:

```
    go connectionHandler(conn, &clients,
            concurrency_chan, &work_wg)
```

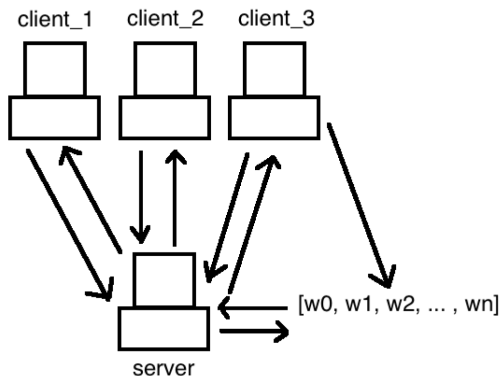
This enables the server to handle multiple clients. In a similar fashion, each client spawns a separate thread to listen for messages from the server. So in each `client.go` instance, the main thread waits for input from standard in, while the secondary thread listens for messages from the server.

```
    go listener(conn)
    for {
        reader := bufio.NewReader(os.Stdin)
        ins, _ := reader.ReadString('\n')
        outs := []byte(ins)
        conn.Write(outs)
    }
```

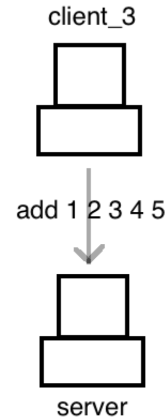
This multithreaded model for the client then enables input while the clients are receiving messages.

3. IMPLEMENTING THE MODEL

The next step is to implement a small interpreted language to enable our distributed computing model. For the intents of this proof-of-concept, the model supports one command, that being the `add`, for addition. An overview of the computational model is below:



A client first sends a command and a work set (w_0 through w_n) to the server. The server then establishes constant contact with the work set (represented by a Go channel in this model), pulling from it and updating it as it sends work to clients and receives results from clients. Finally, the last item in the work set (the final result) is pulled from the work set and sent back to the originating client. In the following, we walk through this model with the accompanying code. First, the client sends the `add` command with what we call a *work set* (w_0 through w_n) on which to perform the work (in this case, `add`).



The server receives the command `add` along with the work set (in this case, 1 2 3 4 5) and recognizes it as a command. Anything not recognized as a command is assumed to be chat text, and is echoed out to all clients.

```
/* split input by spaces into an array */
code:=strings.Split(string(buf[0:n-1])," ")
    if len(buf[0:n]) > 2 {
        /* check the first item in array */
        switch code[0] {
            /* if client sends add cmd */
            case "add":
                /* acknowledge locally */
                fmt.Fprintf(os.Stderr, "add\n")
                /* strip cmd from code */
                code = code[1:]
                /*
                 * write work set
                 * to concurrency channel
                 */
                for _,val := range code {
                    concurrency_chan <- val
                }
                /* start addition thread */
                go net_add(work_wg,
                    concurrency_chan,
                    clients, conn)
```

In the code above, the comments make the execution path clear. The server recognizes the command, strips the command from the input array leaving the work set, adds the work set to the concurrency channel, and starts a thread with the `net_add` function.

```
func net_add(work_wg *sync.WaitGroup,
  concurrency_chan chan string,
  clients *[]net.Conn, conn net.Conn){
   /* while 2+ items in channel to add */
   for len(concurrency_chan) > 1 {
      /* for each client */
      for _, conn := range *clients {
         /* add a thread to waitgroup */
         work_wg.Add(1)
         /* start thread for client */
         go _net_add(work_wg, conn,
          concurrency_chan)
      }
      /* wait for threads to finish */
      work_wg.Wait()
   }
   /* read final answer from channel */
   final_ans := <- concurrency_chan
   /* send final answer back to client */
   conn.Write([]byte(final_ans))
}
```

The flow of the `net_add` function is relatively simple and outlined in the included comments. While there is work to be done (items in `channel > 2`), spawn a client work thread and add a thread to the `waitGroup` for each client, and then wait for all clients to finish their computation (signaled by the `waitGroup`). It would seem that the `waitGroup.done()` signal is missing, but that is included in the server's answer acknowledgment code that is triggered when a client sends back a computed answer.

The spawned `_net_add` function is rather simple:

```
func _net_add(work_wg *sync.WaitGroup,
  conn net.Conn,
  concurrency_chan chan string){
   /* read <num num> from work set */
   val := <- concurrency_chan +
    " " + <-concurrency_chan
   /* send work to client */
   client_cmd := "add " + val
   conn.Write([]byte(client_cmd))
   /* ans handled in connection handler */
}
```

It reads two items from the concurrency channel, prepends the `add` command to them, and sends it to the client it was passed.

Each client:

```
for {
  var inbuf [512]byte
  n, _ := conn.Read(inbuf[0:])
  /* handles add command */
  if string(inbuf[0:3]) == "add" {
    var ins []byte
    for i:=0; i<n; i++ {
      ins = append(ins, inbuf[i])
    }
    in_arr := strings.Split(
      string(ins[:]), " ")
    x, _ := strconv.Atoi(in_arr[1])
    y, _ := strconv.Atoi(in_arr[2])
    /* add work */
    add := (x + y)
    /* prepend a:(nswer) flag */
    add_sendback := ":a "
    add_sendback += strconv.Itoa(add)
    /* send to server */
    conn.Write([]byte(add_sendback))
  }
```

Reads in the message from the server, recognizes the `add` command, performs the necessary conversions to perform integer addition, computes the result, prepends the `:a` flag to the result so the server knows it is receiving the result of work, and sends it back to the server.

The server then:

```
/* handle :a result flag */
if (string(buf[0:2]) == ":a"
  && len(buf) > 2) {
   /* separate answer from :a flag */
   ans_tuple := strings.Split(
    string(buf[:]), " ")
   /* remove null bytes */
   ans_tup := strings.Split(
    ans_tuple[1], "\x00")
   /* write answer to channel */
   concurrency_chan <- ans_tup[0]
   /* signal waitGroup thread done */
   work_wg.Done()
}
```

Reads in the message from the client, recognizes the `:a` result flag, separates the result from the entire message, trims the null bytes, writes the answer to the work set channel, and signals the `waitGroup` that a thread has finished its work. Recall, the `waitGroup` was mentioned in the walkthrough of the `net_add` function. This is where the `waitGroup` is signaled, as a thread has finished its work.

This process continues until there is only one item left in the work set channel, that being the final answer. This answer is then sent to the originating client:

```
for len(concurrency_chan) > 1 {
      ...
      work_wg.Wait()
}
/* read final answer from chan */
final_ans := <- concurrency_chan
/* send final answer back */
conn.Write([]byte(final_ans))
```

And the client prints it to the screen with its normal receive

-> echo behavior:

```
for {
  var inbuf [512]byte//
  n, _ := conn.Read(inbuf[0:])
  /* handle clean disconnect */
  if string(inbuf[0:3]) == "_:q" {
     ...
  } else if string(inbuf[0:3]) == "add" {
     ...
  /* simply echo message */
  } else {
      if n != 0 {
          fmt.Fprintf(os.Stdout,
           "%s\n", inbuf[0:n])
      }
  }
}
```

4. CONCLUSION

Looking back on the project, though it would have been rather interesting to have come across Robert Krimen's Javascript interpreter *Otto* [2] earlier on and to have leveraged it for this project, the purposes of this project were more educational, of which it achieved its aims. The distributed model of binary addition is simple enough to comprehend in a distributed sense for anyone involved, and Go's support for concurrency helps ease the pain of mental mapping of everything 'going on' in a distributed network application.

4.1 Further Extension

Though addition is a simple binary operation, this proof-of-concept paves the way for further extension of the model. By adding new commands to the current set and implementing their interpretation in both client and server, the power of this model can be extended. In addition to building up the power of the model, further efficiency modifications can be made such as condensing verbose lines which would eliminate unnecessary variables. One possible application would be the inclusion of Otto. This would infinitely extend the lingually computational power of the model, as it would be able to leverage Javascript for its computational power.

5. REFERENCES

[1] M. Y. Arslan, I. Singh, S. Singh, H. V. Madhyastha, K. Sundaresan, and S. V. Krishnamurthy. Computing while charging: Building a distributed computing infrastructure using smartphones. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 193–204, New York, NY, 2012. ACM Press.

[2] R. Krimen. A javascript interpreter in Go (golang). Available: https://github.com/robertkrimen/otto [Last accessed: 5 May 2016].

[3] J. Newmarch. Network programming with go. Available: https://jan.newmarch.name/go/ [Last accessed: 5 May 2016], 2012.