C and Golang for distributed programming

Dominik Stecher

This document will provide the reader with an in-depth look into the inner workings and usage of C and Golang for distributed programming using either Sockets or Remote Procedure Calls (RPC). The focus will lie on ease of use, error proneness and usage of advanced programming concepts.

For the first part of this document this paper recaps the usage of sockets in C to establish a low-level implementation baseline by comparing a standard C implementation of sockets to the net-package used by Golang. This will give a first impression of how Golang handles low-level programming which even today is dominated by the C/C++ programming languages due to their widespread use in Operating Systems (OS).

The second part will take a deeper look into advanced concepts of distributed programming, especially RPCs. The paper will compare the official Oracle ONC+ Developer's Guide example for an RPC implementation in C with the Golang example tutorial provided by gRPC. This is to ensure a fair and even playing field for both programming languages without any undue influence from differences in the writer's skill level in either one.

Definition of distributed programming

Distributed programming is required to create software that can run on separate logical (Container, VM) or physical (Multiple server nodes, server-client) infrastructure. This implies a distinct lack of communication between the individual software elements, which cannot exchange data over shared variables and memory like threads or call methods working on those data elements. For this reason, standardized approaches to enable data exchange and communication between two separate pieces of software are required. Fields of application for distributed programming include among others the areas included in the following list.

- Telephone and computer networks
- Web applications
- Games with network or multiplayer options
- Redundant fault tolerant real time systems, i.e. 2oo3
- Automated industrial production lines
- Supercomputer clusters and volunteer computing, i.e. BOINC

In the modern, interconnected world, distributed programming is an essential cornerstone. It is used to ensure that the software everyone in the western world uses on a daily basis in their everyday life, from smartphones to online shopping, works in an efficient, fast and responsive way. In addition, it keeps factories running smoothly and helps with running scientific tasks on supercomputer clusters.

It is for these reasons that most modern programming languages strive to make programming these distributed systems both easier and safer in terms of error handling and programming in the first place. Furthermore, streamlined and more easily readable syntax can speed up development and reduce oversights in the final product.

Programming with sockets

Originally developed and known as Berkeley sockets in the 1980s, sockets have become a well-known and - as POSIX sockets - universal Application Programming Interface (API) provided by all major Operating Systems (OS). Due to the length of their existence, their popularity and versatility, sockets can be seen as the foundation of distributed programming as they provide the necessary connectivity for distributed software to communicate and interact.

The sources for this chapter are the Unix socket server example from tutorialspoint.com as well as the "Build a concurrent TCP server in Go" from opensource.com, written by Mihalis Tsoukalos. Both are minimalist implementations making them suitable candidates for a direct comparison. The Golang implementation is capable of running on both Windows and Linux systems whereas the C implementation only runs on Linux/Unix there will be mentions on what is necessary for a Windows implementation.

Header files and Packages

The `net`-Package of Golang provides a streamlined interface for socket usage while still allowing access to low-level variables and functions when necessary. Furthermore the `net`-Package can be used for both Windows and Linux/Unix, whereas C requires different header files depending on the OS. The package provides support for TCP and UDP sockets as well as IPv4 and IPv6. All other packages are not required to use sockets but provide functionality for the rest of the code.

Go

```
import (
    "bufio"
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

For C a variety of libraries are required or recommended in order to provide the necessary support for different OSs. To make this a fair comparison the necessary header files for Windows and Linux/Unix will be used to ensure that the example software will run on both systems.

C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef _WIN32
/* Header files for Windows */
#include <winsock2.h>

#else
/* Header files for UNIX/Linux */
#include <netinet/in.h>
#include <netdb.h>
#endif
```

After the first two include statements for `stdio.h` and `stdlib.h`, `#ifdef` is required to differentiate between possible OSs, which adds a layer of potential errors to the source code.

For Windows-code, the `winsock2.h` is required. The previous version, `winsock.h` is still available and can be found in tutorials and books but using `winsock.h` can lead to issues. Since some compilers or more complicated projects require `windows.h` to be included, using `winsock.h` can lead to compiler errors, as `windows.h`, when compiling for modern Windows versions such as Windows XP, Vista, 7, 8, 10, already includes `winsock2.h`. This can be avoided by using either identical versions of winsock or by including `winsock.h` after `winsock2.h` has already been included, as `winsock2.h` will prevent the compiler from including `winsock.h`.

`netinet/in.h` adds type definitions for IPv4 and IPv6 internet adresses. `netdb.h` provides functions such as `gethostbyaddr` and `gethostbyname`.

The C approach has the advantage of better modularity as it allows to include small sets of required libraries with narrowly defined functionality such as `sys/socket.h`, `arpa/inet.h`. However, this adds complexity which can lead to confusion as to which library provides what exact functionality. This effort is doubled by having to maintain two separate lists of libraries for different OSs. The Golang approach to reduce complexity is already apparent in the way the entire socket and network functionality is concentrated in one package. In addition to that, the net-Package itself simplifies many functions as will be shown in the following chapters.

Socket initialization and binding

Before a socket can be used to transmit and receive data it must first be initialized to define its behaviour as well as bound to a host address for the host OS to redirect data to it and make it visible and usable from outside the program itself.

Go

```
arguments := os.Args
PORT := ":" + arguments[1]
l, err := net.Listen("tcp4", PORT)
if err != nil {
        fmt.Println(err)
        return
}
```

The `net.Listen()` function both initializes the socket and binds it to a network port, which, in this case, is passed through the command line. `net.Listen()` accepts two `string` parameters. The first determines the network type for initialization and must be one of the following: `tcp`, `tcp4`, `tcp6`, `unix` or `unixpacket`. The second parameter defines the port the socket will be bound to. In case the address is left empty or set to `0` the function will automatically select a port number or, for TCP networks, listen for unicast and anycast IP adresses. `net.Listen()` utilizes the ability of Golang to return more than one value by returning a `Listener` and an `error`.

C

```c
int sockfd, newsockfd, portno, clilen;
char buffer[256];
struct sockaddr_in serv_addr, cli_addr;
int n, pid;

/* Socket creation */
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0) {
      perror("ERROR opening socket");
      exit(1);
}

/* Socket initialization */
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = 5001;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

/* Socket binding */
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
      perror("ERROR on binding");
      exit(1);
   }
/* Socket listening */
listen(sockfd,5);
```

In C, the functionality of the `net.Listen()` function is split into four steps, creation, initialization, binding and listening. This process includes redundancy such as the first parameter of the `socket()` function and the `serv_addr.sin_family` attribute. In addition, it requires two individual tests, whether or not an error occurred at socket creation and socket binding. In Golang this is handled by the net-Package internally and the error is returned as an individual return value of the correct type `error`, not as an integer value. The same applies for the socket itself, which, in Linux/Unix, is an `int` whereas Golang uses `listener` to properly identify the return value and to provide better type safety. The port to which the socket will be bound is of type `int` as well, but has to be converted to network byte order using the `htons()` function.

From a programming perspective, the C approach requires knowledge of which variable holds what value, which functions can return an error instead of the expected value in the same return value as well as hard to debug issues such as using network byte order for certain variables. For code maintenance this means more lines of code to understand while at the same time being less intuitive to read, increasing the necessary time budget. The Golang code can be written in a very descriptive way, enabling the programmer to understand the code by simply reading it.

To give an example, we want to change from IPv4 to IPv6. For Golang, the only change is to turn `tcp4` into `tcp6` as the number indicates the internet protocol version. The property is stored in one

variable in the code and even for an inexperienced programmer the deduction from `tcp` followed by one of the two internet protocol versions is easy to make. For C, the first thing to change is `sockaddr_in` to `sockaddr_in6`, as the `sin_addr` struct only holds 4 byte IPv4 addresses in an unsigned 32bit integer. The `in6_addr` struct is composed of an `unsigned char` array of length 16, which renders any mathematical operations on IPv4 addresses incompatible, increasing the maintenance overhead. In addition to that, during creation and initialization the parameter or variable `AF_INET` must be changed to `AF_INET6`. In all those cases, the lack of a `4` in `sockaddr_in`, `sin_addr` and `AF_INET` makes it not immediately obvious that a change is required, increasing error potential though oversight.

C

```
/* IPv4 struct */
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};

/* IPv4 implementation */
struct in_addr {
    uint32_t s_addr; // 32bit or 4byte
};

/* IPv6 implementation */
struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

If the C code is supposed to run on both Linux/Unix and Windows, code complexity increases even further. Using the Windows socket library changes the data type of sockets from `int` to `SOCKET`. The socket initialization as well as closing the socket requires further Windows specific code. Depending on the code, even more changes can be required as the listed ones represent the minimum changes necessary. The fact that the entire Golang code so far requires less lines of code than the Windows specific initalization shows the advantages of Golang as a relatively new programming language adapted to current programming needs whereas C shows signs of being an older language, which grew and had to adapt to new circumstances while maintaining backwards compatibility.

C

```c
/* Creating variable before creation to account for different data type */
#ifdef _WIN32
    SOCKET sockfd;
#else
    int sockfd;
#endif

/* Windows specific initialization */
#ifdef _WIN32
    /* TCP socket initialization with winsock library */
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD (1, 1);
    if (WSAStartup (wVersionRequested, &wsaData) != 0){
      perror("ERROR on initialization");
      exit(1);
      }
#endif

/* Different commands to close the socket  */
#ifdef _WIN32
        closesocket(sockfd);
#else
        close(sockfd);
#endif
```

Using a Socket to communicate

After setting up a socket it can be used to accept a connection and communicate. To keep the initial socket open for further requests a separation between the new connection and the initial socket, which is used to handle requests.

Go

```
defer l.Close()

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
}
```

The `defer` statement replaces Javas `finally` as it delays - or defers - the `Close()` to the time when the encapsulating function, in this case the `main()` function terminates. It does not matter if the termination stems from an error and resulting panic or if it is a regular `return` statement. The unbounded `for` loop keeps the connection open and the server running to accept new connections.

The `Accept()` function returns a new connection of type `net.Conn` and an error, separating the new connection from the initial socket listener `l`. After checking for potential errors the connection is handed over to a go routine for further processing. This go routine can `read`, `write` and `close` the connection on its own.

Go routines have the advantage of not requiring a new thread or process to execute, instead they are concurrent, lightweight routines, scheduled and multiplexed on already existing threads by a Go runtime scheduler. The advantage over threads is that the OS does not see individual Go routines and does not need to restore any registers when switching between Go routines, unlike threads. Go routines start with only 2kB of memory whereas threads will consume at least 1MB of memory to account for the required stack. Last but not least it cuts down on OS calls to create new processes or threads. Those advantages make Golang an ideal language for web servers for example, as those usually handle a large amount of connections simultaneously.

C

```c
/* listen() is included again to provide better readability and context */
listen(sockfd,5);
clilen = sizeof(cli_addr);
   /* infinite loop to serve more than one request */
   while (1) {
      newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);

      if (newsockfd < 0) {
         perror("ERROR on accept");
         exit(1);
      }

      /* Create child process */
      pid = fork();

      if (pid < 0) {
         perror("ERROR on fork");
         exit(1);
      }

      if (pid == 0) {
         /* Client process */
         close(sockfd);
         doprocessing(newsockfd);
         exit(0);
      }
      else {
         close(newsockfd);
      }
   }
```

In C, the `while(1)` loop ensures a continued service. Within the loop the connection is accepted and made available under a new socket. After checking for errors a child process is created using the system call `fork()` and after checking for errors caused by fork, the original connection is closed and the child process is started. The child process uses the newly created socket for the connection. Both the newly created socket and the child process are terminated by the `exit()` function or, in case of a failed `fork()` that was not detected by the error handling, the `close()` function is used on `newsockfd` to avoid a resource leak.

In addition to consuming more memory and CPU time as indicated in the Golang section, error handling is necessary in two separate locations as well as a potential resource leak by leaving a socket open. Golangs `defer` in combination with using `net.Conn` connections instead of new sockets avoids this problem in a way that makes the code both safer and easier to understand.

Programming with gRPC

RPC is a more elaborate concept compared to sockets as it allows the client to directly call a procedure or function on the server. While one could create a similar behaviour using sockets, RPC frameworks allow for a standardized and transparent software. RPC frameworks do require additional software packages to provide this functionality and, for the gRPC framework, further plugins such as Google Protobuf are required.

Creating an RPC Server

Using Golang, the first step is to create a `.proto` file. This file will be compiled using the aforementioned Google Protobuf compiler and contains the service offered as well as the RPC-capable functions. In typical Golang fashion the code is easily readable and self explanatory.

Go

```
service RouteGuide {
  rpc GetFeature(Point) returns (Feature) {}
  ...
}
```

After the `.proto` file is compiled a `.pb.go` file is generated which contains an interface for the new server as well as declarations of the RPC functions. These declarations must be turned into definitions to provide full functionality. Note that transfer parameters as well as return values are already provided.

Go

```
type routeGuideServer struct {
        /* server properties */
}
...

func (s *routeGuideServer) GetFeature(ctx context.Context, point *pb.Point)
(*pb.Feature, error) {
        /* body of the GetFeature function must be added here */
}
...
```

Setting up the actual server uses the `net` package from the previous chapter to create a TCP socket, then set up and start the server by creating a new server with `grpc.NewServer()`, registering the new server and starting the server using the previously created socket.

Go

```
lis, err := net.Listen("tcp", fmt.Sprintf(":%d", *port))
if err != nil {
        log.Fatalf("failed to listen: %v", err)
}
grpcServer := grpc.NewServer()
pb.RegisterRouteGuideServer(grpcServer, &routeGuideServer{})
... // determine whether to use TLS
grpcServer.Serve(lis)
```

With C and `rpcgen` writing the RPC declaration of a function is more complex. Procedures or functions are part of a program and receive an identification number, in this case `0x20000001`. In addition, similar to Golang, the data types of transfer parameters as well as the return value, are declared.

C

```
/* msg.x: Remote msg printing protocol */
 program MESSAGEPROG {
     version PRINTMESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
     } = 1;
} = 0x20000001;
```

The implementation uses a header file generated by `rpcgen`. The RPC function gains an additional argument containing the invocation context, in this case `req`. Both arguments and return values are pointers instead of the values. Finally, the function name from the RPC declaration is converted to all lower-case letters and the version is added, here `_1`.

C

```
#include <stdio.h>
#include "msg.h"                /* msg.h generated by rpcgen */

int *
printmessage_1(msg, req)
    char **msg;
    struct svc_req *req;        /* details of call */
{
    static int result;          /* must be static! */

    ...

    result = 1;
    return (&result);
}
```

Creating an RPC Client

The Golang RPC client setup works similar to the server setup but with client specific functions such as creating a connection to the server instead of setting up a TCP socket using `grpc.Dial()` and handing the connection over to the new client function `pb.NewRouteGuideClient()`. After that, all RPC functions are available using the `client`. The connection is terminated automatically when `defer conn.Close()` is triggered on exiting the surrounding function.

Go

```
/* Create connection to server */
conn, err := grpc.Dial(*serverAddr)
if err != nil {
    ...
}
defer conn.Close()
/* Create client */
client := pb.NewRouteGuideClient(conn)
/* Call function using client */
feature, err := client.GetFeature(context.Background(), &pb.Point{409146138,
-746188906})
if err != nil {
        ...
}
```

The RPC client in C is a close analogon to the Golang code with the main differences being the more complex `clnt_create` command, more complex error checking due to the usage of pointers and the need to explicitly call `clnt_destroy()` to avoid resource leaks.

C

```c
/* Create client handle */
  clnt = clnt_create(server, MESSAGEPROG,   PRINTMESSAGEVERS,   "visible");
    if (clnt == (CLIENT *)NULL) {
        clnt_pcreateerror(server);
        exit(1);
    }
/* Call RPC function */
    result = printmessage_1(&message, clnt);
    if (result == (int *)NULL) {
        clnt_perror(clnt, server);
        exit(1);
    }
/* Check for error */
    if (*result == 0) {
        fprintf(stderr, "%s: could not print your message\n",argv[0]);
        exit(1);
    }
  /* Delete client */
  clnt_destroy( clnt );
```

Conclusion

Overall, C and Golang can provide the same basic functionality for distributed programming. C also has the advantage of many compilers which support a large variety of CPU architectures and of which some are certified for safety critical use cases. This is relevant for the industry 4.0 and Internet of Things (IoT) boom as well as industry applications. Aside from that, Golang is superior in terms of ease of use, development time and support for more complex programming concepts. This is no surprise however, considering that C, released in 1972, is 37 years older than Golang. This means Golang has had the opportunity to be adapted and developed specifically for modern requirements and use cases.

First, unlike C, Golang can return more than one value. This allows Golang code to return a separate error value of type `error`, making error handling more easily readable and type safe compared to C, where, depending on the original return value the error check must use either a test for `x == -1` or `x == NULL`, if the return value is not a number. It also reduces the potential for unchecked errors as the second return value serves as a reminder that an error may occur and be returned. Finally, Golang requires less error handling in general when compared to C as the expanded functionality of Golang packages compared to C libraries reduces the number of functions called by the programmer, concentrating error handling in one function call. There are multiple examples of C code given where error handling takes up more lines of code than the actual code itself.

Second, the syntax of Golang is more precise and more easily readable. This is very pronounced when comparing sockets with attributes such as `AF_INET`, which do not follow common network terminology as does the somewhat equal `tcp4` argument in Golang but also do not indicate that there are different versions available such as `AF_INET6` or `tcp6` respectively. Programmers without prior knowledge of these acronyms are forced to look up their meaning as well as derived versions, making code maintenance more time consuming.

Third, with Golang, parameters are concentrated in one location in the code. This results in less potential for errors when upgrading C code, as parameters in C may require changes in multiple locations, such as when changing from IPv4 to IPv6, which requires multiple parameter changes as well as using different structs to account for the increased length of IPv6 addresses.

Fourth, cross platform development is easier using Golang as the packages offer enough of an abstraction layer that differences in the OS are no longer relevant. This stands in stark contrast to C, which, as shown in the socket example, requires different libraries and different functions, data types and code segments without a counterpart when using a different OS. This makes C code more convoluted and less readable compared to Golang and maintenance more complicated as two separate implementations for the different OSs must be changed in addition to checking the shared code segments for compatibility with the changes.

Fifth, the RPC support of Golang is more automated and powerful than C offerings. This begins with easier RPC function declaration avoiding a special syntax or manual numbering and version keeping and extends to easy support for modern network standards such as `TLS` and data streams. It should be mentioned that the gRPC framework, used for the Golang code samples, supports C++, which can offer an upgrade path for legacy code.

Sixth, Golang uses a stronger type concept compared to C, as demonstrated with sockets and child processes both being of type `int` and error messages hidden as a `-1` return value in these cases. Golang uses specific data types such as `Conn`, `TCPConn`, `Listener`, `TCPListener`, `error` and `TCPAddr`. This allows the use of automated systems such as type checking at compile time to detect errors. This is a similar problem to the use of `#define` in C when used incorrectly for constants.

Seventh and last but not least, Golang offers go routines, which are lightweight replacements for threads and enable fast, safe and easy concurrent programming, making it ideal for the dynamic and fast changing world of web service development. It allows programmers to write concurrent code faster while also reducing the hardware requirements to run the resulting software. Considering that web services are usually expected to serve a large amount of clients at the same time this makes Golang a prime candidate for such projects.

Lastly, as a footnote, it should be mentioned that Golang is capable of using C code embedded in the Golang code. In the case of requiring the low level capabilities of C, this can be used while simultaneously maintaining the advantages of Golang for other code segments.