ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 18 Loose Coupling

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 23, 2024

Outline

Reading Assignment

- This lecture: 8
- Next Lecture: Apache Kafka
  `https://kafka.apache.org/documentation/#design`

Outline

Tight Coupling

Communication Patterns

Coupling

- Degree of direct knowledge between components.
  - E.g. a client that sends requests to a service.
  - Cannot be avoided for a system to function.
- Tightly coupled: a great deal of knowledge.
  - E.g. to require same version of shared library.
  - An easy choice for short-term.
  - Problematic for long-term evolutions – one must change all tightly coupled components at the same time.
- Loosely coupled: minimal direct knowledge.
  - Components are relatively independent, interacting through mechanisms that are stable and mature.
  - Require more up-front planning but easier to upgrade or even be rewritten, without affecting existing systems.

Forms of Tight Coupling

- Things that are wrongly assumed to not change.
  - Modern software engineering practices are based on the assumption that requirements will change frquently.
- Fragile exchange protocols
  - Clients and servers communicating via SOAP/XML messages rely on strict formats that cannot be updated independently.
  - REST messages have less coupling because both clients and servers may choose to ignore attributes they don't understand.
- Shared dependencies
  - Require to use specific libraries and even specific versions of libraries for communication, e.g. Java RMI.
- Shared point-in-time
  - A request-response messaging creates coupling in time as the service must be available at the time.
  - A bad choice if users are not waiting for immediate answers.
- Fixed addresses
  - Have you ever hardcoded a file path to read data from?
  - Network services may relocate, and having multiple of them helps to separate production, testing, and development.

Outline

Tight Coupling

Communication Patterns

Communications Between Services

- Via message passing
  - Shared memory communications are less popular nowadays among servers as they make communication implicit and thus prevent optimizations toward delays and failures.
- Make use of a contract.
  - Backward-compatible with existing components.
  - Forward-compatible with future components.
- Messaging patterns
  - Request-response (synchronous): requester (client) issues a request to a receiver (server) and waits for a response.
  - Publish-subscribe (asynchronous): publisher send a message to a middleware (event bus, message exchange, etc.) and subscribers pick it up later.
- We will focus on request-response for this lecture and leave publish-subscribe to the next.

Request-Response Messaging

- A layered approach where structures can be introduced
  - TCP/UDP: messages in bytes, need to handle message length for TCP, and ordering and retrying for UDP.
  - Remote procedure calls (RPC): use messages to provide illusions to call a function on another server by sending function name and parameters and receiving returned values.
  - HTTP: messages as text, e.g. HTML, XML, json.
  - REST: messages in json to represent complex data.
  - GraphQL: json as a query language.
- Synchronous communications like request-response are easy to reason and straightforward to implement.
  - Point-to-point
  - Responses are either available or not, indicating failures that can be handled further.
- Not ideal for one-to-many communications or when requester needs to wait for long time.

HTTP Requests in Go

```
// Get issues a GET to the specified URL
func Get(url string) (*http.Response, error)
// Post issues a POST to the specified URL
func Post(url, contentType string, body io.Reader) (*Response, error)
type Response struct {
  Status     string // e.g. "200 OK"
  StatusCode int    // e.g. 200
  // Header maps header keys to values.
  Header Header
  // Body represents the response body.
  Body io.ReadCloser
  // ContentLength records the length of the associated content. The
  // value -1 indicates that the length is unknown.
  ContentLength int64
  // Request is the request that was sent to obtain this Response.
  Request *Request
}
```

- From the `net/http` package.
- Provide convenience functions like `Get` and `Post`.
  - That one can call directly without the need to create some objects for the request first.

HTTP GET Example

```go
package main
import (
  "fmt"
  "io"
  "net/http"
)
func main() {
  resp, err := http.Get("http://example.com") // Send an HTTP GET
  if err != nil {
    panic(err)
  }
  defer resp.Body.Close()                     // Close your response!
  body, err := io.ReadAll(resp.Body)          // Read body as []byte
  if err != nil {
    panic(err)
  }
  fmt.Println(string(body))
}
```

HTTP POST Example

```
package main
import (
  "fmt"
  "io"
  "net/http"
  "strings"
)
const json = `{ "name":"Matt", "age":44 }`       // This is our JSON
func main() {
  in := strings.NewReader(json)                  // Wrap JSON with an io.Reader
  // Issue HTTP POST, declaring our content-type as "text/json"
  resp, err := http.Post("http://example.com/upload", "text/json", in)
  if err != nil {
    panic(err)
  }
  defer resp.Body.Close()                        // Close your response!
  message, err := io.ReadAll(resp.Body)
  if err != nil {
    panic(err)
  }
  fmt.Printf(string(message))
}
```

Remote Procedure Calls (RPC) with gRPC

- gRPC is a full-featured data exchange framework.
  - Open sourced in 2015 by Google, and with CNCF from 2017.
  - A modern RPC solution as an alternative to RESTful services.
- Advantages
  - Conciseness: more compact than json, less network I/O.
  - Speed: binary format is much faster to produce and consume.
  - Strong-typing: avoid conversions, easier for troubleshooting.
  - Feature-rich: e.g. authentication, encryption, timeout, and compression.
- Disadvantages
  - Contract-driven: more coupling, less suitable for external facing services.
  - Binary format: not human-readable, complicating troubleshooting.

gRPC Message Definition

```
syntax = "proto3";
option go_package = "github.com/cloud-native-go/ch08/keyvalue";
message GetRequest {
  string key = 1;
}
message GetResponse {
  string value = 1;
}
message PutRequest {
  string key = 1;
  string value = 2;
}
message PutResponse {}
message DeleteRequest {
  string key = 1;
}
message DeleteResponse {}
```

- Make use of protocol buffers, fairly straightforward to follow.
- The protocol compiler generates code for clients and servers.
  - Available for most programming languages.

gRPC Service Definition

```
service KeyValue {
  rpc Get(GetRequest) returns (GetResponse);
  rpc Put(PutRequest) returns (PutResponse);
  rpc Delete(DeleteRequest) returns (DeleteResponse);
}
```

- A service consists of a group of methods.
- Define an interface without providing implementations.
    - Methods are used in the client program with your choice of programming language.
    - Methods are implemented in the server program with your choice of programming language.
    - Clients and servers can use different programming languages.

Implementing gRPC Server

```
// generated server interface to be implemented
type KeyValueServer interface {
  Get(context.Context, *GetRequest) (*GetResponse, error)
  Put(context.Context, *PutRequest) (*PutResponse, error)
  Delete(context.Context, *DeleteRequest) (*PutResponse, error)
}

// server.go
... // package, import etc.
type server struct {
  pb.UnimplementedKeyValueServer // embed the generated struct
}
func (s *server) Get(ctx context.Context, r *pb.GetRequest) (*pb.GetResponse, e
  log.Printf("Received GET key=%v", r.Key)
  value, err := Get(r.Key)
  return &pb.GetResponse{Value: value}, err
}
... // Put, Delete, etc.
```

Implementing gRPC Server (cont.)

```
...
func main() {
  // Create a gRPC server and register our KeyValueServer with it
  s := grpc.NewServer()
  pb.RegisterKeyValueServer(s, &server{})
  // Open a listening port on 50051
  lis, err := net.Listen("tcp", ":50051")
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
  // Start accepting connections on the listening port
  if err := s.Serve(lis); err != nil {
    log.Fatalf("failed to serve: %v", err)
  }
}
```

Implementing gRPC Client

```
// generated client interface to be used
type KeyValueClient interface {
  Get(ctx context.Context, in *GetRequest, opts ...grpc.CallOption) (*GetRespon
  Put(ctx context.Context, in *PutRequest, opts ...grpc.CallOption) (*PutRespon
  Delete(ctx context.Context, in *DeleteRequest, opts ...grpc.CallOption) (*Put
}

// client.go
... // package, import etc.
func main() {
  // Set up a connection to the gRPC server
  conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure(),
    grpc.WithBlock(), grpc.WithTimeout(time.Second))
  ... // error handling
  defer conn.Close()
  // Get a new instance of our client
  client := pb.NewKeyValueClient(conn)
  ...
```

Implementing gRPC Client (cont.)

```
  ...
  var action, key, value string
  if len(os.Args) > 2 {
    action, key = os.Args[1], os.Args[2]
    value = strings.Join(os.Args[3:], " ")
  }
  // Use context to establish a 1-second timeout.
  ctx, cancel := context.WithTimeout(context.Background(), time.Second)
  defer cancel()
  switch action {
  case "get":
    r, err := client.Get(ctx, &pb.GetRequest{Key: key})
    ... // error handling
    log.Printf("Get %s returns: %s", key, r.Value)
  case "put":
    _, err := client.Put(ctx, &pb.PutRequest{Key: key, Value: value})
    ... // error handling
    log.Printf("Put %s", key)
  default:
    log.Fatalf("Syntax: go run [get|put] KEY VALUE...")
  }
}
```

Summary

- Coupling is unavoidable.
- But we can keep it minimal with a good choice of communication patterns.