

Go + Flutter Course

Communication & Microservices

Timur Harin

Lecture 06: Communication & Microservices

Building scalable, distributed, real-time applications

Block 6: Real-time & Microservices

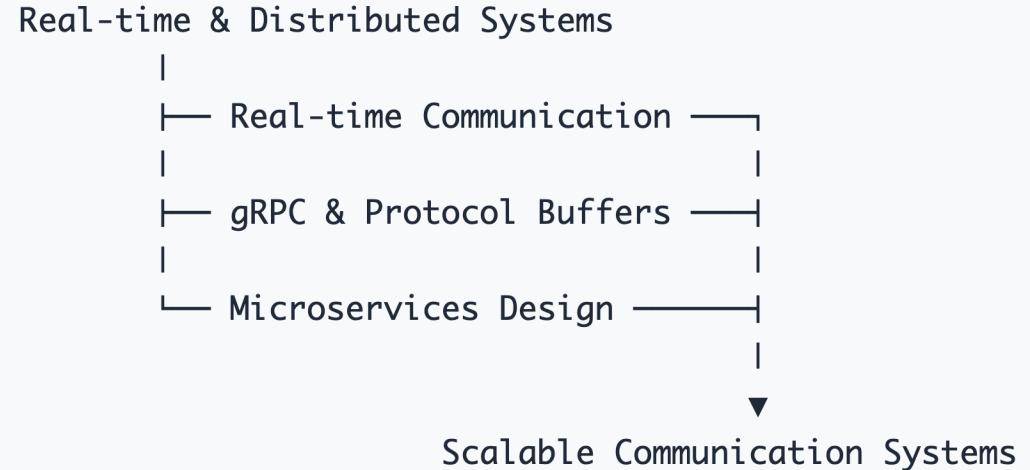
Lecture 06 Overview

- **Real-time Communication:** WebSockets, WebRTC, VoIP, and peer-to-peer systems
- **gRPC & Protocol Buffers:** High-performance RPC communication
- **Microservices Architecture:** Design patterns and communication strategies

What we'll learn

- Why real-time communication matters in modern apps
- WebSocket implementation in Go and Flutter
- WebRTC for peer-to-peer audio/video communication
- VoIP systems with SIP protocol and audio processing
- gRPC service development and client integration
- Microservices patterns and anti-patterns
- Service discovery and inter-service communication

Learning path



- **Foundation:** Real-time communication protocols and patterns
- **High-performance:** gRPC for efficient service-to-service communication
- **Architecture:** Microservices design and communication strategies
- **Integration:** Cross-platform mobile and server communication

Part I: Real-time Communication

Real-time communication enables instant data exchange between clients and servers, creating responsive and interactive user experiences.

Why do we need different communication protocols?

- **HTTP Request/Response:** Good for traditional web pages, APIs
- **Real-time needs:** Chat, live updates, gaming require instant communication
- **Network constraints:** Mobile networks, firewalls, NAT traversal

Common real-time scenarios

- Traditional HTTP: Client → Request → Server → Response → Client (high latency)
- WebSocket: Client ↔ Persistent Connection ↔ Server (instant bidirectional)
- Server-Sent Events: Client ← Stream of Updates ← Server (server-to-client only)

When to use WebSocket vs alternatives

WebSocket is perfect for:

- **Chat applications**: Instant messaging, group conversations
- **Live gaming**: Real-time multiplayer interactions
- **Collaborative editing**: Google Docs-style simultaneous editing
- **Trading platforms**: Real-time stock price updates
- **Live sports scores**: Instant score updates during matches

WebSocket vs other solutions:

Scenario	HTTP Polling	Server-Sent Events	WebSocket	Best Choice
Chat app	✗ Inefficient	✗ One-way only	✓ Perfect	WebSocket
Live news feed	✗ Delays	✓ Efficient	✓ Overkill	SSE
API calls	✓ Simple	✗ Wrong tool	✗ Overkill	HTTP
Gaming	✗ Too slow	✗ One-way	✓ Essential	WebSocket
Live charts	✗ Inefficient	✓ Good enough	✓ Better	WebSocket

HTTP vs WebSocket

HTTP Polling (inefficient)

```
# Request every 2 seconds
GET /api/messages?since=1642781234 HTTP/1.1
Host: example.com
Authorization: Bearer token123
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 45
```

```
{"messages": [], "has_new": false}
```

- Empty responses waste bandwidth
- 2-second delay for new messages
- HTTP headers overhead (200+ bytes)
- Resources wasted on empty polls

WebSocket (efficient)

```
# Initial handshake only
GET /chat HTTP/1.1
Upgrade: websocket
Connection: Upgrade
```

```
# Then instant messages:
{"type": "message", "text": "Hello!", "user": "John"}
{"type": "message", "text": "Hi there!", "user": "Jane"}
```

- Instant delivery (0ms delay)
- Only 50 bytes per message
- No HTTP header overhead
- Bidirectional communication
- Single persistent connection

Simple WebSocket server example

Basic connection handling

```
import (
    "github.com/gorilla/websocket"
    "net/http"
)

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true // Allow all origins
    },
}

func handleWebSocket(w http.ResponseWriter, r *http.Request) {
    // Upgrade HTTP to WebSocket
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        return
    }
    defer conn.Close()
    // ... connection handling code ...
}
```

Message broadcasting

```
type Message struct { // Simple message structure
    Type   string `json:"type"`
    Content string `json:"content"`
    User    string `json:"user"`
} // Broadcast to all connected clients
var clients = make(map[*websocket.Conn]bool)
func broadcastMessage(message Message) {
    for client := range clients {
        err := client.WriteJSON(message)
        if err != nil {
            client.Close()
            delete(clients, client)
        }
    }
}
func handleMessage(conn *websocket.Conn) {
    var msg Message
    err := conn.ReadJSON(&msg)
    if err == nil {
        broadcastMessage(msg) // Send to all clients
    }
}
```

Flutter WebSocket client example

Simple connection setup

```
class SimpleWebSocket {
  WebSocket? _socket;
  final _messageController = StreamController<String>.broadcast();
  Stream<String> get messages => _messageController.stream;
  Future<void> connect(String url) async {
    _socket = await WebSocket.connect(url);
    _socket!.listen((data) {
      final message = jsonDecode(data);
      _messageController.add('${message['user']}: ${message['content']}');
    });
  }

  void sendMessage(String content) {
    final message = {
      'type': 'message',
      'content': content,
      'user': 'CurrentUser'
    };
    _socket?.add(jsonEncode(message));
  }

  void dispose() {
    _messageController.close();
    _socket?.close();
  }
}
```

Receiving messages in Flutter widget

```
class _ChatWidgetState extends State<ChatWidget> {
  final _webSocket = SimpleWebSocket();
  final _messages = <String>[];
  late StreamSubscription<String> _subscription;
  @override
  void initState() { // Will be called when the widget is created
    super.initState();
    _webSocket.connect('ws://localhost:8080/chat').then((_) {
      _subscription = _webSocket.messages.listen((msg) { // init subscription
        setState(() {
          _messages.add(msg);
        });
      });
    });
  }
  void _sendMessage(String text) {
    _webSocket.sendMessage(text);
    setState(() {
      _messages.add('You: $text');
    });
  }
  @override
  void dispose() {
    _subscription.cancel();
    _webSocket.dispose();
    super.dispose();
  }
  // ... UI code to display _messages ...
}
```

HTTP vs gRPC: size comparison

```
POST /api/v1/users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Accept: application/json
User-Agent: MyApp/1.0
Content-Length: 156
{
  "name": "John Doe",
  "email": "john.doe@example.com",
  "age": 30,
  "city": "New York",
  "country": "USA",
  "phone": "+1-555-123-4567"
}
```

Total size: ~650 bytes

- Headers: ~480 bytes
- JSON body: ~156 bytes
- Human readable but verbose

```
// user.proto
message CreateUserRequest {
    string name = 1;           // "John Doe"
    string email = 2;          // "john.doe@example.com"
    int32 age = 3;             // 30
    string city = 4;           // "New York"
    string country = 5;         // "USA"
    string phone = 6;           // "+1-555-123-4567"
}
```

Binary encoding:

```
0A 08 4A 6F 68 6E 20 44 6F 65 12 15 6A 6F 68 6E 2E 64 6F 65...
```

Total size: ~85 bytes

- HTTP/2 headers: ~20 bytes
- Protocol Buffer body: ~65 bytes
- **87% smaller than JSON!**

JSON-RPC: Simple RPC over HTTP

JSON-RPC is a lightweight Remote Procedure Call protocol using JSON for data exchange. Simpler than gRPC but less efficient.

```
{
  // Request
  "jsonrpc": "2.0",
  "method": "createUser",
  "params": {
    "name": "John Doe",
    "email": "john@example.com"
  },
  "id": 1
}
// Response
{
  "jsonrpc": "2.0",
  "result": {
    "id": 12345,
    "name": "John Doe",
    "email": "john@example.com",
    "created_at": 1642781234
  },
  "id": 1
}
```

```

type UserService struct{}
type CreateUserParams struct {
  Name string `json:"name"`
  Email string `json:"email"`
}
type User struct {
  ID      int   `json:"id"`
  Name    string `json:"name"`
  Email   string `json:"email"`
  CreatedAt int64 `json:"created_at"`
}
func (s *UserService) CreateUser(params CreateUserParams) (*User, error) {
  if params.Name == "" {
    return nil, fmt.Errorf("name is required")
  }
  user := &User{
    ID:      generateID(),
    Name:    params.Name,
    Email:   params.Email,
    CreatedAt: time.Now().Unix(),
  }
  return user, nil
}
// ... JSON-RPC handler setup ...

```

WebRTC: Real-time Peer-to-Peer

What is WebRTC?

- **WebRTC (Web Real-Time Communication)** is a technology that enables direct, real-time audio, video, and data communication between browsers and mobile apps.
- Built into all major browsers and supported by many mobile SDKs.
- Designed for **low-latency, peer-to-peer** connections without requiring plugins or external software.

How does WebRTC work?

- **Peer-to-peer:** Connects users directly, minimizing server load and latency.
- **Media streams:** Supports audio, video, and arbitrary data channels.
- **NAT traversal:** Uses ICE, STUN, and TURN protocols to connect users behind firewalls and NATs.
- **Encryption:** All streams are encrypted end-to-end by default.
- **Signaling:** Requires a signaling server (using WebSocket, HTTP, etc.) to exchange connection info, but media/data flows directly between peers.

WebRTC: Use Cases and Advantages

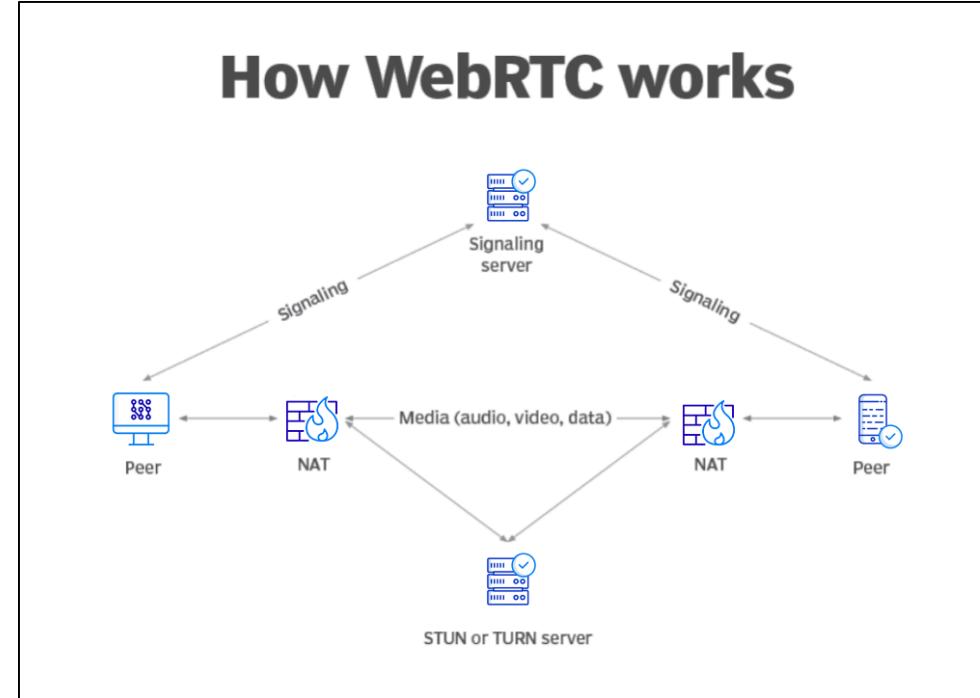
What is WebRTC used for?

- **Video calls:** Zoom, Google Meet, WhatsApp, and many others use WebRTC for real-time video.
- **Voice calls:** High-quality, low-latency audio communication.
- **Screen sharing:** Share desktop or app windows in real time.
- **File transfer:** Send files directly between users without uploading to a server.
- **Gaming:** Real-time multiplayer games with direct peer-to-peer data channels.

Why choose WebRTC over other protocols?

- **Direct peer-to-peer:** No server bottleneck for data, reducing latency and cost.
- **Media optimized:** Built-in support for adaptive bitrate, echo cancellation, and network resilience.
- **Security:** End-to-end encryption is mandatory.
- **Cross-platform:** Works in browsers, mobile apps, and even IoT devices.
- **Scalability:** Offloads bandwidth from servers, making it ideal for large-scale video/voice applications.

When to use WebRTC?



- When you need **real-time, interactive audio/video** between users.
- When **server bandwidth/cost** is a concern (media doesn't go through your server).
- When **privacy** and **encryption** are required by design.
- For **direct file/data transfer** between users without storing on a server.
- For **low-latency, high-quality** communication (e.g., live collaboration, remote control, telemedicine).

Protocol comparison summary

Choosing the right protocol for your use case:

Protocol	Best For	Pros	Cons	Size Efficiency
HTTP/JSON	REST APIs, Web apps	Simple, debuggable	Verbose, slower	★★
WebSocket	Real-time chat, gaming	Bidirectional, instant	More complex	★★★
gRPC	Microservices, high-performance	Fast, type-safe, streaming	Learning curve	★★★★★
JSON-RPC	Simple RPC calls	Easy to implement	Limited features	★★★
WebRTC	Video/audio calls	P2P, media streaming	Complex setup	★★★★

Example of usage patterns

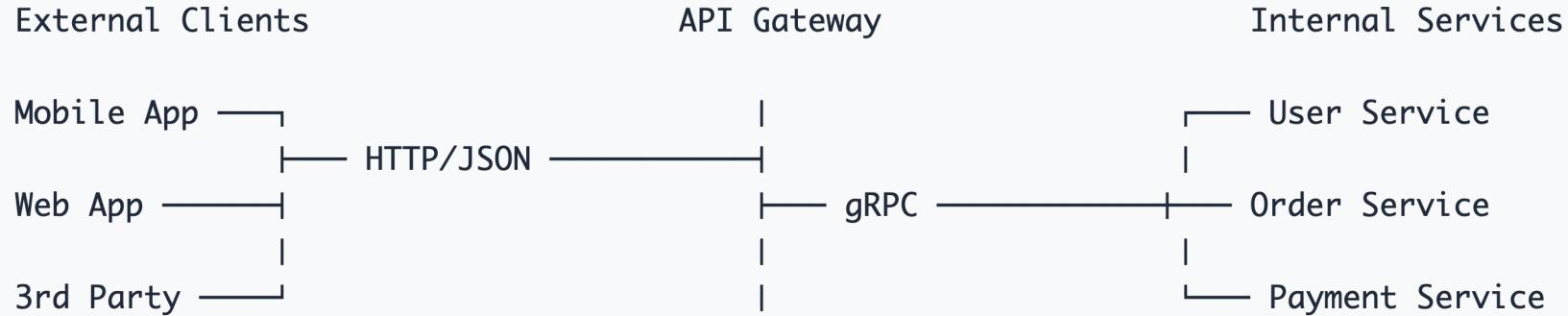
Real-world usage patterns:



Performance comparison (1000 requests):

- **HTTP/JSON**: ~650KB total
- **gRPC**: ~85KB total (**87% less bandwidth**)
- **WebSocket**: ~50KB total (after connection)

Internal vs External API architecture



External API requirements (HTTP/REST): Internal API requirements (gRPC):

- **Browser compatibility:** Native fetch API, no special libraries
- **Tooling:** Extensive ecosystem (Swagger, OpenAPI, etc.)
- **Caching:** HTTP caching with CDNs and proxies
- **Firewall friendly:** Port 80/443, well-known protocol

- **Performance:** 7x faster than JSON, binary Protocol Buffers
- **Type safety:** Generated clients prevent API misuse
- **Streaming:** Real-time data exchange between services
- **Error handling:** Rich error codes and structured error details

When to use gRPC vs HTTP APIs

Decision matrix for API protocol selection:

Use Case	gRPC	HTTP/REST	Reason
Mobile app → API Gateway	✗	✓	Browser/mobile compatibility
Web frontend → Backend	✗	✓	JavaScript fetch API, debugging
Service → Service	✓	✗	Performance, type safety
Real-time updates	✓	✗	Bidirectional streaming
High-frequency calls	✓	✗	Lower latency, smaller payload
3rd party integrations	✗	✓	Industry standard, tooling
Public API	✗	✓	Documentation, accessibility

Simple gRPC service example

Protocol definition

```
// calculator.proto
syntax = "proto3";
package calculator;

service Calculator {
    rpc Add(Numbers) returns (Result);
    rpc Multiply(Numbers) returns (Result);
}

message Numbers {
    double a = 1;
    double b = 2;
}

message Result {
    double value = 1;
}
```

Go server implementation

```
type calculatorServer struct {
    pb.UnimplementedCalculatorServer
}

func (s *calculatorServer) Add(ctx context.Context, req *pb.Numbers) (*pb.Result, error) {
    result := req.A + req.B
    return &pb.Result{Value: result}, nil
}

func (s *calculatorServer) Multiply(ctx context.Context, req *pb.Numbers) (*pb.Result, error) {
    result := req.A * req.B
    return &pb.Result{Value: result}, nil
}

func main() {
    lis, _ := net.Listen("tcp", ":50051")
    server := grpc.NewServer()
    pb.RegisterCalculatorServer(server, &calculatorServer{})
    server.Serve(lis)
}
```

Generate code:

```
protoc --go_out=. --go-grpc_out=. calculator.proto
```

gRPC streaming example

Server streaming proto

```
service NumberService {
    // Server streaming: send multiple responses
    rpc GetPrimes(PrimeRequest) returns (stream Prime);

    // Client streaming: receive multiple requests
    rpc SumNumbers(stream Number) returns (Sum);

    // Bidirectional streaming
    rpc Chat(stream ChatMessage) returns (stream ChatMessage);
}

message PrimeRequest {
    int32 limit = 1;
}

message Prime {
    int32 value = 1;
}
```

Server streaming implementation

```
func (s *numberServer) GetPrimes(req *pb.PrimeRequest, stream pb.NumberService_GetPrimesServer) error {
    for i := 2; i <= int(req.Limit); i++ {
        if isPrime(i) {
            prime := &pb.Prime{Value: int32(i)}
            if err := stream.Send(prime); err != nil {
                return err
            }

            // Send one prime every 100ms
            time.Sleep(100 * time.Millisecond)
        }
    }
    return nil
}

// Client usage:
stream, _ := client.GetPrimes(ctx, &pb.PrimeRequest{Limit: 100})
for {
    prime, err := stream.Recv()
    if err == io.EOF {
        break
    }
    fmt.Printf("Prime: %d\n", prime.Value)
}
```

User Service with gRPC

```
// user_service.proto
syntax = "proto3";
package user;

import "google/protobuf/timestamp.proto";
import "google/protobuf/empty.proto";

service UserService {
    rpc CreateUser(CreateUserRequest) returns (User);
    rpc GetUser(GetUserRequest) returns (User);
    rpc UpdateUser(UpdateUserRequest) returns (User);
    rpc DeleteUser(DeleteUserRequest) returns (google.protobuf.Empty);
    rpc ListUsers(ListUsersRequest) returns (stream User);
    rpc ValidateUser(ValidateUserRequest) returns (UserValidation);
}

message User {
    string id = 1;
    string name = 2;
    string email = 3;
    string phone = 4;
    bool active = 5;
    google.protobuf.Timestamp created_at = 6;
    google.protobuf.Timestamp updated_at = 7;
}
```

```
message CreateUserRequest {
    string name = 1;
    string email = 2;
    string phone = 3;
}

message ValidateUserRequest {
    string user_id = 1;
}

message UserValidation {
    bool is_valid = 1;
    string reason = 2;
}
```

User service with gRPC

```

package user

import (
    "context"
    "database/sql"
    "time"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "google.golang.org/protobuf/types/known/timestamppb"
    pb "github.com/example/user/proto"
)

type UserServer struct {
    pb.UnimplementedUserServiceServer
    db *sql.DB
}

func (s *UserServer) CreateUser(ctx context.Context, req *pb.CreateUserRequest) (*pb.User, error) {
    // Validate input
    if req.Name == "" {
        return nil, status.Errorf(codes.InvalidArgument, "name is required")
    }
    if req.Email == "" {
        return nil, status.Errorf(codes.InvalidArgument, "email is required")
    }

    // Check if user exists
    var exists bool
    err := s.db.QueryRow("SELECT EXISTS(SELECT 1 FROM users WHERE email = $1)", req.Email).Scan(&exists)
    if err != nil {
        return nil, status.Errorf(codes.Internal, "database error: %v", err)
    }
}

```

```

if exists {
    return nil, status.Errorf(codes.AlreadyExists, "user with email %s already exists", req.Email)
}

// Create user
user := &pb.User{
    Id:    generateUUID(),
    Name:  req.Name,
    Email: req.Email,
    Phone: req.Phone,
    Active: true,
    CreatedAt: timestamppb.Now(),
    UpdatedAt: timestamppb.Now(),
}

// Save to database
_, err = s.db.Exec(`INSERT INTO users (id, name, email, phone, active, created_at, updated_at)
VALUES ($1, $2, $3, $4, $5, $6, $7)`,
    user.Id, user.Name, user.Email, user.Phone, user.Active,
    user.CreatedAt.AsTime(), user.UpdatedAt.AsTime())
if err != nil {
    return nil, status.Errorf(codes.Internal, "failed to create user: %v", err)
}

return user, nil
}

```

Notification Service with streaming

```
// notification_service.proto
syntax = "proto3";
package notification;

service NotificationService {
    // Server streaming - send notifications to clients
    rpc Subscribe(SubscribeRequest) returns (stream Notification);

    // Send notification to specific user
    rpc SendNotification(SendNotificationRequest) returns (google.protobuf.Empty);

    // Broadcast to all users
    rpc Broadcast(BroadcastRequest) returns (google.protobuf.Empty);
}

message Notification {
    string id = 1;
    string user_id = 2;
    NotificationType type = 3;
    string title = 4;
    string message = 5;
    map<string, string> data = 6;
    google.protobuf.Timestamp created_at = 7;
}
```

```
enum NotificationType {
    INFO = 0;
    WARNING = 1;
    ERROR = 2;
    SUCCESS = 3;
    ORDER_UPDATE = 4;
    USER_MESSAGE = 5;
}

message SubscribeRequest {
    string user_id = 1;
    repeated NotificationType types = 2;
}

message SendNotificationRequest {
    string user_id = 1;
    NotificationType type = 2;
    string title = 3;
    string message = 4;
    map<string, string> data = 5;
}
```

Notification service with streaming

```

package notification

import (
    "context"
    "sync"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    pb "github.com/example/notification/proto"
)

type NotificationServer struct {
    pb.UnimplementedNotificationServiceServer
    subscribers map[string][]pb.NotificationService_SubscribeServer
    mutex       sync.RWMutex
}

func (s *NotificationServer) Subscribe(req *pb.SubscribeRequest, stream pb.NotificationService_SubscribeServer) error {
    // Add subscriber
    s.mutex.Lock()
    if s.subscribers[req.UserId] == nil {
        s.subscribers[req.UserId] = make([]pb.NotificationService_SubscribeServer, 0)
    }
    s.subscribers[req.UserId] = append(s.subscribers[req.UserId], stream)
    s.mutex.Unlock()

    // Send welcome notification
    welcomeNotification := &pb.Notification{
        Id:      generateNotificationID(),
        UserId: req.UserId,
        Type:   pb.NotificationType_INFO,
        Title:  "Connected",
        Message: "You are now connected to notifications",
        CreatedAt: timestamppb.Now(),
    }
}

func (s *NotificationServer) SendNotification(ctx context.Context, req *pb.SendNotificationRequest) (*emptypb.Empty, error) {
    if err := stream.Send(welcomeNotification); err != nil {
        return err
    }

    // Keep connection alive until client disconnects
    <-stream.Context().Done()

    // Remove subscriber
    s.removeSubscriber(req.UserId, stream)
    return nil
}

func (s *NotificationServer) SendNotification(ctx context.Context, req *pb.SendNotificationRequest) (*emptypb.Empty, error) {
    notification := &pb.Notification{
        Id:      generateNotificationID(),
        UserId: req.UserId,
        Type:   req.Type,
        Title:  req.Title,
        Message: req.Message,
        Data:   req.Data,
        CreatedAt: timestamppb.Now(),
    }

    s.mutex.RLock()
    subscribers := s.subscribers[req.UserId]
    s.mutex.RUnlock()

    // Send to all user's active connections
    for _, stream := range subscribers {
        go func(stream pb.NotificationService_SubscribeServer) {
            if err := stream.Send(notification); err != nil {
                // Remove failed stream
                s.removeSubscriber(req.UserId, stream)
            }
        }(stream)
    }
}

return &emptypb.Empty{}, nil
}

```

gRPC middleware and interceptors

Authentication interceptor

```
func AuthInterceptor(authService AuthService) grpc.UnaryServerInterceptor {
    return func(ctx context.Context, req interface{},
        info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
        // Skip auth for certain methods
        if isPublicMethod(info.FullMethod) {
            return handler(ctx, req)
        }
        // Extract metadata
        md, ok := metadata.FromIncomingContext(ctx)
        if !ok {
            return nil, status.Errorf(codes.Unauthenticated, "missing metadata")
        }
        // Get authorization header
        authHeaders := md.Get("authorization")
        if len(authHeaders) == 0 {
            return nil, status.Errorf(codes.Unauthenticated, "missing authorization token")
        }
        token := strings.TrimPrefix(authHeaders[0], "Bearer ")
        // Validate token
        userID, err := authService.ValidateToken(token)
        if err != nil {
            return nil, status.Errorf(codes.Unauthenticated, "invalid token: %v", err)
        }
        // Add user ID to context
        ctx = context.WithValue(ctx, "user_id", userID)
        return handler(ctx, req)
    }
}
```

Logging and metrics interceptor

```
var (
    grpcRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "grpc_requests_total",
            Help: "Total number of gRPC requests",
        },
        []string{"method", "status"})
    grpcRequestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name: "grpc_request_duration_seconds",
            Help: "gRPC request duration in seconds",
        },
        []string{"method"})
)

func LoggingInterceptor(ctx context.Context, req interface{},
    info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
    start := time.Now()
    log.Printf("gRPC request started: %s", info.FullMethod)
    resp, err := handler(ctx, req)
    duration := time.Since(start)
    statusCode := status.Code(err)
    log.Printf("gRPC request completed: %s, duration: %v, status: %s",
        info.FullMethod, duration, statusCode)
    // Record metrics
    grpcRequestsTotal.WithLabelValues(info.FullMethod, statusCode.String()).Inc()
    grpcRequestDuration.WithLabelValues(info.FullMethod).Observe(duration.Seconds())
    return resp, err
}
```

Why split your application?

The monolith problem:

Single Large Application

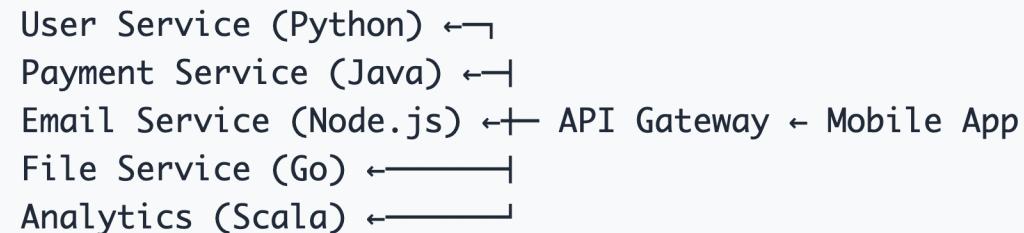
- └─ User Management (Team A needs Python)
- └─ Payment Processing (Team B needs Java)
- └─ Email Service (Team C needs Node.js)
- └─ File Upload (Team D needs Go)
- └─ Analytics (Team E needs Scala)

Problems:

- One technology stack for everything
- Deploy entire app for small changes
- One team's bug affects everyone
- Difficult to scale individual features

The microservices solution

The microservices solution:



Benefits:

- Each team picks best technology
- Independent deployments
- Failure isolation
- Scale services individually

Databases for microservices

Database per Service pattern

User Service ↔ PostgreSQL (Users DB)
Order Service ↔ MongoDB (Orders DB)
Product Service ↔ Elasticsearch (Products DB)
Analytics Service ↔ ClickHouse (Analytics DB)

Benefits:

- Service autonomy and independence
- Technology diversity (choose best DB for use case)
- Data isolation and security
- Independent scaling

Challenges:

- Cross-service queries become complex
- Data consistency across services
- Transaction management
- Data duplication

Data consistency patterns

Saga Pattern (Distributed Transactions) Example: Order processing saga

```
type OrderSaga struct {
    steps []SagaStep
}
type SagaStep struct {
    Execute func() error
    Compensate func() error
    Executed bool
}
func (s *OrderSaga) Execute() error {
    for i, step := range s.steps {
        if err := step.Execute(); err != nil {
            s.rollback(i) // Compensate all executed steps
            return err
        }
        s.steps[i].Executed = true
    }
    return nil
}
func (s *OrderSaga) rollback(failedStep int) {
    for i := failedStep - 1; i >= 0; i-- {
        if s.steps[i].Executed {
            s.steps[i].Compensate()
        }
    }
}
```

```
func CreateOrderSaga(orderID, userID string, items []Item) *OrderSaga {
    return &OrderSaga{
        steps: []SagaStep{
            {
                Execute: func() error {
                    return reserveInventory(items)
                },
                Compensate: func() error {
                    return releaseInventory(items)
                },
            },
            {
                Execute: func() error {
                    return chargePayment(userID, total)
                },
                Compensate: func() error {
                    return refundPayment(userID, total)
                },
            },
            // ...
        },
    }
}
```

Event Sourcing pattern

Traditional state storage

```
// Store current state only
type User struct {
    ID      string `json:"id"`
    Name    string `json:"name"`
    Email   string `json:"email"`
    Status  string `json:"status"`
    Balance float64 `json:"balance"`
}

// Lost information:
// - When was user created?
// - How many times email changed?
// - Balance history?
// - Who made changes?
```

Event sourcing approach

```
type Event struct {
    ID      string      `json:"id"`
    Type    string      `json:"type"`
    Data    interface{} `json:"data"`
    Timestamp time.Time `json:"timestamp"`
    UserID   string      `json:"user_id"`
}

type UserCreated struct {
    UserID string `json:"user_id"`
    Name   string `json:"name"`
    Email  string `json:"email"`
}

type EmailChanged struct {
    UserID  string `json:"user_id"`
    OldEmail string `json:"old_email"`
    NewEmail string `json:"new_email"`
}
```

Event Sourcing pattern

Rebuilding state from events

```
func (u *User) ApplyEvent(event Event) {  
    switch event.Type {  
        case "UserCreated":  
            data := event.Data.(UserCreated)  
            u.ID = data.UserID  
            u.Name = data.Name  
            u.Email = data.Email  
            u.Status = "active"  
            u.Balance = 0.0  
  
        case "EmailChanged":  
            data := event.Data.(EmailChanged)  
            u.Email = data.NewEmail  
  
        case "BalanceUpdated":  
            data := event.Data.(BalanceUpdated)  
            u.Balance = data.NewBalance  
    }  
}
```

```
func GetUserState(userID string) (*User, error) {  
    events, err := getEventsByUserID(userID)  
    if err != nil {  
        return nil, err  
    }  
  
    user := &User{}  
    for _, event := range events {  
        user.ApplyEvent(event)  
    }  
  
    return user, nil  
}
```

CQRS

Separation of reads and writes

```
// Command side (writes)
type UserCommandHandler struct {
    eventStore EventStore
}

func (h *UserCommandHandler) CreateUser(cmd CreateUserCommand) error {
    // Validate command
    if cmd.Email == "" {
        return errors.New("email required")
    }
    // Create event
    event := Event{
        Type: "UserCreated",
        Data: UserCreated{
            UserID: cmd.UserID,
            Name: cmd.Name,
            Email: cmd.Email,
        },
        Timestamp: time.Now(),
    }
    // Store event
    return h.eventStore.SaveEvent(event)
}
```

```
// (Command Query Responsibility Segregation)

func (h *UserCommandHandler) UpdateEmail(cmd UpdateEmailCommand) error {
    // Business logic validation
    if !isValidEmail(cmd.NewEmail) {
        return errors.New("invalid email")
    }

    event := Event{
        Type: "EmailChanged",
        Data: EmailChanged{
            UserID: cmd.UserID,
            OldEmail: cmd.OldEmail,
            NewEmail: cmd.NewEmail,
        },
        Timestamp: time.Now(),
    }

    return h.eventStore.SaveEvent(event)
}
```

CQRS

Query side (reads)

```
// Query side (reads) - optimized for queries
type UserQueryHandler struct {
    readDB ReadDatabase // Could be MongoDB, Elasticsearch, etc.
}

type UserView struct {
    ID      string `json:"id"`
    Name    string `json:"name"`
    Email   string `json:"email"`
    Status  string `json:"status"`
    Balance float64 `json:"balance"`
    LastActivity time.Time `json:"last_activity"`
    TotalOrders int     `json:"total_orders"`
    // Denormalized data for fast queries
}
```

```
func (h *UserQueryHandler) GetUser(userID string) (*UserView, error) {
    return h.readDB.FindUserByID(userID)
}

func (h *UserQueryHandler) SearchUsers(query string) ([]*UserView, error) {
    return h.readDB.SearchUsers(query)
}

// Event handler updates read models
func (h *UserQueryHandler) HandleUserCreated(event Event) {
    data := event.Data.(UserCreated)
    userView := &UserView{
        ID:          data.UserID,
        Name:        data.Name,
        Email:       data.Email,
        Status:      "active",
        Balance:     0.0,
        LastActivity: event.Timestamp,
    }
    h.readDB.SaveUser(userView)
}
```

API versioning strategies

URL versioning

```
// v1/users endpoint
func (h *UserHandler) GetUserV1(w http.ResponseWriter, r *http.Request) {
    user := &UserV1{
        ID:      "123",
        Name:    "John Doe",
        Email:   "john@example.com",
    }
    json.NewEncoder(w).Encode(user)
}

// v2/users endpoint (added phone field)
func (h *UserHandler) GetUserV2(w http.ResponseWriter, r *http.Request) {
    user := &UserV2{
        ID:      "123",
        Name:    "John Doe",
        Email:   "john@example.com",
        Phone:   "+1-555-1234", // New field
    }
    json.NewEncoder(w).Encode(user)
}

// Routing
mux.HandleFunc("/api/v1/users/{id}", h.GetUserV1)
mux.HandleFunc("/api/v2/users/{id}", h.GetUserV2)
```

Header versioning

```
func (h *UserHandler) GetUser(w http.ResponseWriter, r *http.Request) {
    version := r.Header.Get("API-Version")

    switch version {
    case "v1", "1.0":
        h.GetUserV1(w, r)
    case "v2", "2.0":
        h.GetUserV2(w, r)
    default:
        h.GetUserLatest(w, r)
    }
}

// Client usage:
req.Header.Set("API-Version", "v2")
```

gRPC versioning

```
// user_v1.proto
syntax = "proto3";
package user.v1;

service UserService {
    rpc GetUser(GetUserRequest) returns (User);
}

// user_v2.proto
syntax = "proto3";
package user.v2;

service UserService {
    rpc GetUser(GetUserRequest) returns (User);
    rpc GetUserProfile(GetUserProfileRequest)
        returns (UserProfile); // New method
}
```

```
// Client-side usage example (Go)

import (
    userv1 "github.com/example/project/proto/user/v1"
    userv2 "github.com/example/project/proto/user/v2"
    "google.golang.org/grpc"
)

func main() {
    conn, _ := grpc.Dial("user-service:50051", grpc.WithInsecure())
    defer conn.Close()

    // Use v1 client
    clientV1 := userv1.NewUserServiceClient(conn)
    user, _ := clientV1.GetUser(ctx, &userv1.GetUserRequest{Id: "123"})

    // Use v2 client for new features
    clientV2 := userv2.NewUserServiceClient(conn)
    profile, _ := clientV2.GetUserProfile(ctx,
        &userv2.GetUserProfileRequest{Id: "123"})
}
```

Backward compatibility strategies

Additive changes (safe)

```
// V1 - Original
type UserV1 struct {
    ID   string `json:"id"`
    Name string `json:"name"`
    Email string `json:"email"`
}

// V2 - Add optional fields (backward compatible)
type UserV2 struct {
    ID       string `json:"id"`
    Name    string `json:"name"`
    Email   string `json:"email"`
    Phone   *string `json:"phone,omitempty"` // Optional
    Avatar  *string `json:"avatar,omitempty"` // Optional
}

// V1 clients ignore new fields, still work
```

Non-breaking API evolution

```
func (h *UserHandler) CreateUser(w http.ResponseWriter, r *http.Request) {
    var req CreateUserRequest
    json.NewDecoder(r.Body).Decode(&req)

    // Handle both old and new request formats
    user := &User{
        ID:   generateID(),
        Name: req.Name,
        Email: req.Email,
    }

    // Optional fields (V2+)
    if req.Phone != "" {
        user.Phone = &req.Phone
    }
    if req.Avatar != "" {
        user.Avatar = &req.Avatar
    }

    // Return compatible response
    json.NewEncoder(w).Encode(user)
}
```

Backward compatibility strategies

Breaking changes handling

```
// Deprecation strategy
type DeprecatedUserResponse struct {
    ID      string `json:"id"`
    Name    string `json:"name"`
    Email   string `json:"email"`
    Deprecated bool `json:"_deprecated"` // Warning flag
    NewAPIURL string `json:"_new_api_url"`
}

func (h *UserHandler) GetUserDeprecated(w http.ResponseWriter, r *http.Request) {
    // Add deprecation warning header
    w.Header().Set("Deprecation", "version=v1")
    w.Header().Set("Sunset", "2024-12-31")
    w.Header().Set("Link", "</api/v2/users>; rel=\"successor-version\"")

    user := &DeprecatedUserResponse{
        ID:      "123",
        Name:    "John Doe",
        Email:   "john@example.com",
        Deprecated: true,
        NewAPIURL: "/api/v2/users/123",
    }

    json.NewEncoder(w).Encode(user)
}
```

Version support matrix

Version	Status	End of Life	Features
v1	Deprecated	2024-12-31	Basic CRUD
v2	Supported	2025-12-31	+ Phone, Avatar
v3	Current	-	+ Social login
v4	Beta	-	+ Multi-tenancy

What we've learned about connections

1. **HTTP/JSON**: Simple, widely supported, and easy to debug, but inefficient for real-time or high-frequency communication due to overhead and latency.
2. **WebSocket**: Enables instant, bidirectional communication over a single persistent connection—ideal for chat, gaming, and live updates.
3. **gRPC**: High-performance, type-safe, binary protocol for service-to-service communication; supports streaming and is much more efficient than JSON over HTTP.
4. **WebRTC**: Peer-to-peer protocol for real-time audio, video, and data; offloads bandwidth from servers and provides end-to-end encryption by default.
5. **Protocol selection**: Choose the right protocol for your use case—HTTP for APIs, WebSocket for real-time, gRPC for microservices, WebRTC for direct media/data between users.
6. **Hybrid architectures**: Combine protocols (e.g., HTTP for external clients, gRPC/WebSocket/WebRTC internally) for optimal performance and scalability.
7. **Versioning and compatibility**: Use versioning strategies (URL, header, proto package) to evolve APIs and maintain backward compatibility.

Thank You!

What's Next:

- Final Project: Build a complete microservices application

Resources:

- gRPC Documentation: <https://grpc.io/docs/>
- WebSocket RFC: <https://tools.ietf.org/html/rfc6455>
- Microservices Patterns: <https://microservices.io/patterns/>
- Course Repository: <https://github.com/timur-harin/sum25-go-flutter-course>

Contact:

- Email: timur.harin@mail.com
- Telegram: @timur_harin

Questions?