gRPC Go

GothamGo 2015

Sameer Ajmani
Tech Lead Manager, Go team, Google

Video

This talk was presented at GothamGo in New York City, October 2015.

Watch the talk on YouTube (https://www.youtube.com/watch?v=vTIyz2QfExc&index=7&list=PLeGxIOPLk9ELh9tsPZMzau6CzMjfMzp9-)

RPC isn't just Remote Procedure Call

In Go, an RPC **starts a goroutine** running on the server and provides *message passing* between the client and server goroutines.

Unary RPC: the client sends a *request* to the server, then the server sends a *response*.

Streaming RPC: the client and server may each send one or more messages.

An RPC ends when:

- both sides are done sending messages
- either side disconnects
- the RPC is canceled or times out

This talk will show how we connect RPCs and streams with goroutines and channels.

Unary RPC: one request, one response

Example: a mobile Maps app requests a route from point A to point B.

On the client side, an RPC blocks until it's done or canceled.

A client uses multiple goroutines to run many RPCs simultaneously.

Each RPC is an exchange between a client goroutine and a server goroutine.

Streaming RPC provides bidirectional message-passing

A client starts a stream with a server.

Messages sent on a stream are delivered FIFO.

Many streams can run simultaneously between the same client and server.

The transport provides buffering and flow control.

Examples:

- bidirectional stream: chat session
- server → client stream: stock ticker
- client → server stream: sensor aggregation

gRPC is a new RPC system from Google

grpc.io (http://grpc.io)

Provides RPC and streaming RPC

Ten languages: **C**, **Java**, **Go**, C++, Node.js, Python, Ruby, Objective-C, PHP, and C#
IDL: **Proto3**
Transport: **HTTP2**

golang.org/x/net/context (http://golang.org/x/net/context) for deadlines, cancelation, and request-scoped values
golang.org/x/net/trace (http://golang.org/x/net/trace) for real-time request traces and connection logging

gRPC users

150+ imports of google.golang.org/grpc (https://godoc.org/google.golang.org/grpc?importers) on godoc.org (http://godoc.org)

- Apcera/Kurma (https://github.com/apcera/kurma): container OS
- Bazil (http://bazil.org): distributed file system
- CoreOS/Etcd (http://coreos.com/etcd/): distributed consistent key-value store
- Google Cloud Bigtable (https://godoc.org/google.golang.org/cloud/bigtable): sparse table storage
- Monetas/Bitmessage (https://github.com/monetas/bmd): transaction platform
- Pachyderm (http://www.pachyderm.io/): containerized data analytics
- YouTube/Vitess (http://vitess.io/): storage platform for scaling MySQL

Demos and Code: Google search

Protocol definition

```
syntax = "proto3";

service Google {
  // Search returns a Google search result for the query.
  rpc Search(Request) returns (Result) {
  }
}

message Request {
  string query = 1;
}

message Result {
  string title = 1;
  string url = 2;
  string snippet = 3;
}
```

Generated code

```
protoc ./search.proto --go_out=plugins=grpc:.
```

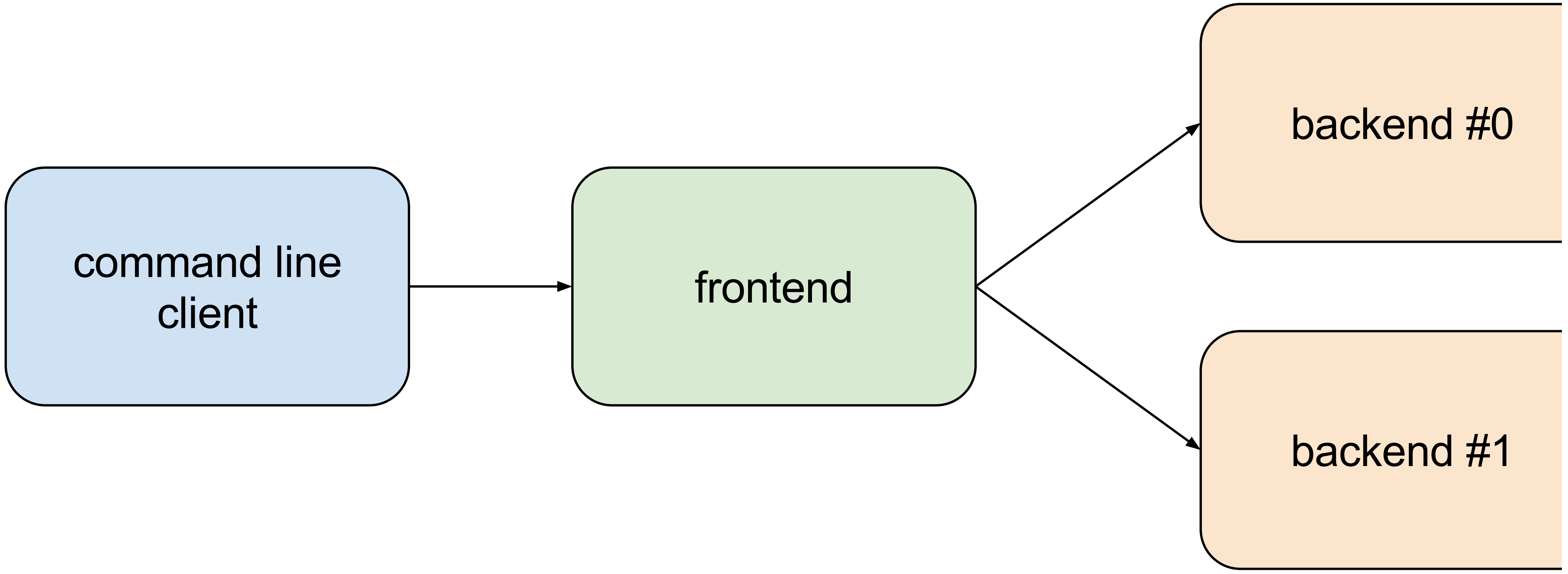
```
type GoogleClient interface {
    // Search returns a Google search result for the query.
    Search(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Result, error)
}
```

```
type GoogleServer interface {
    // Search returns a Google search result for the query.
    Search(context.Context, *Request) (*Result, error)
}
```

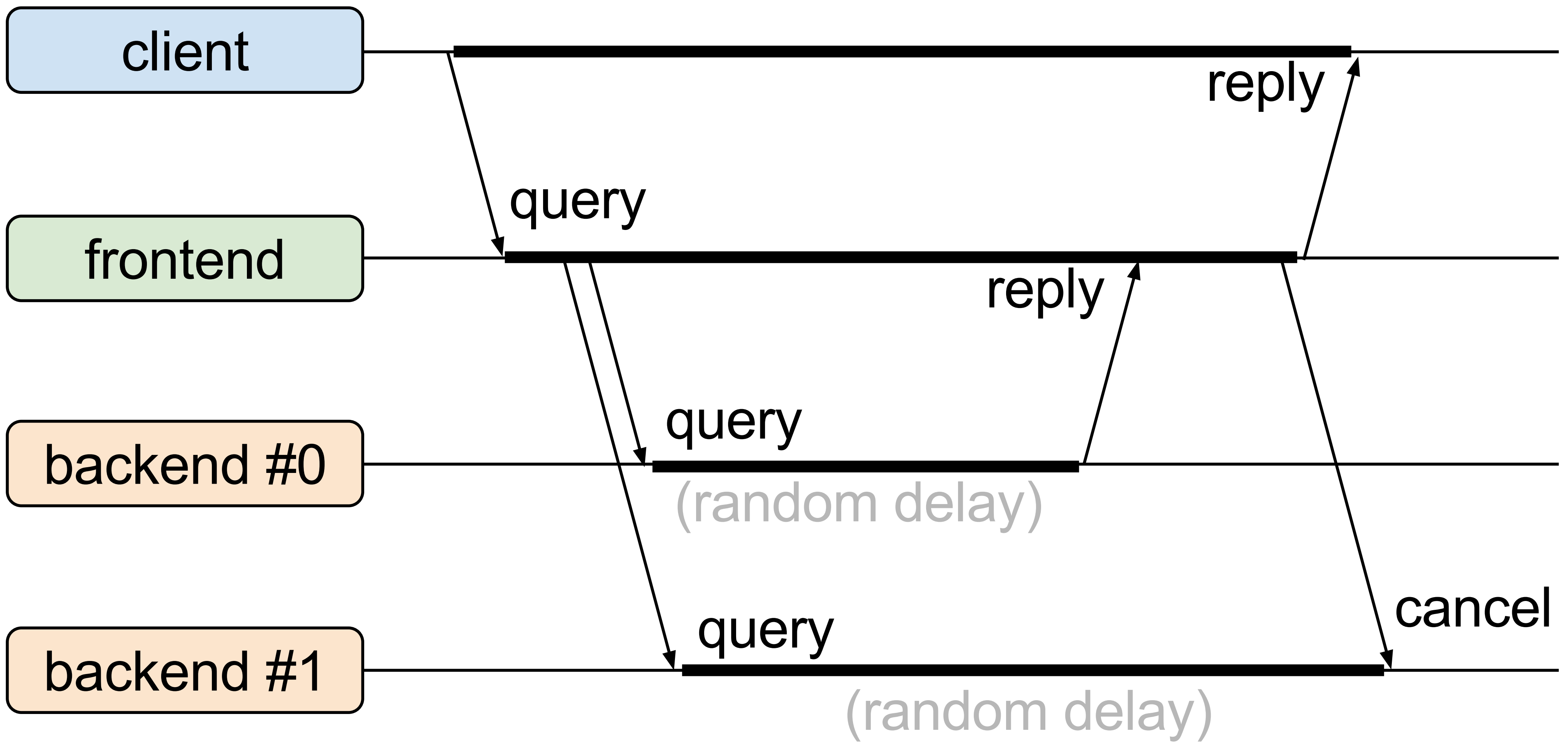
```
type Request struct {
    Query string `protobuf:"bytes,1,opt,name=query" json:"query,omitempty"`
}
```

```
type Result struct {
    Title   string `protobuf:"bytes,1,opt,name=title" json:"title,omitempty"`
    Url     string `protobuf:"bytes,2,opt,name=url" json:"url,omitempty"`
    Snippet string `protobuf:"bytes,3,opt,name=snippet" json:"snippet,omitempty"`
}
```

System diagram



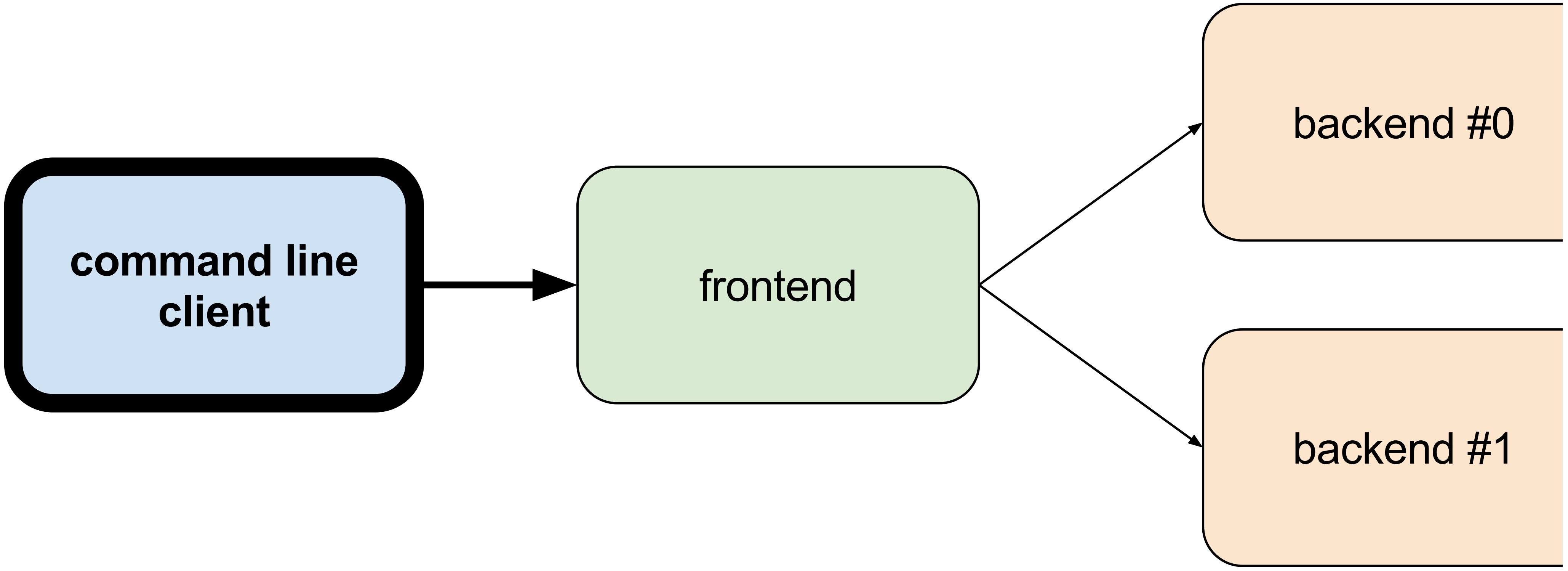
Frontend runs Search on both backends and returns first result



Demo client --mode=search

- Frontend request traces
- Backend request traces
- Connection event logs

Client code



Client code (main)

```
import pb "golang.org/x/talks/2015/gotham-grpc/search"
```

```
func main() {
    flag.Parse()

    // Connect to the server.
    conn, err := grpc.Dial(*server, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("fail to dial: %v", err)
    }
    defer conn.Close()
    client := pb.NewGoogleClient(conn)

    // Run the RPC.
    switch *mode {
    case "search":
        search(client, *query)
    case "watch":
        watch(client, *query)
    default:
        log.Fatalf("unknown mode: %q", *mode)
    }
}
```

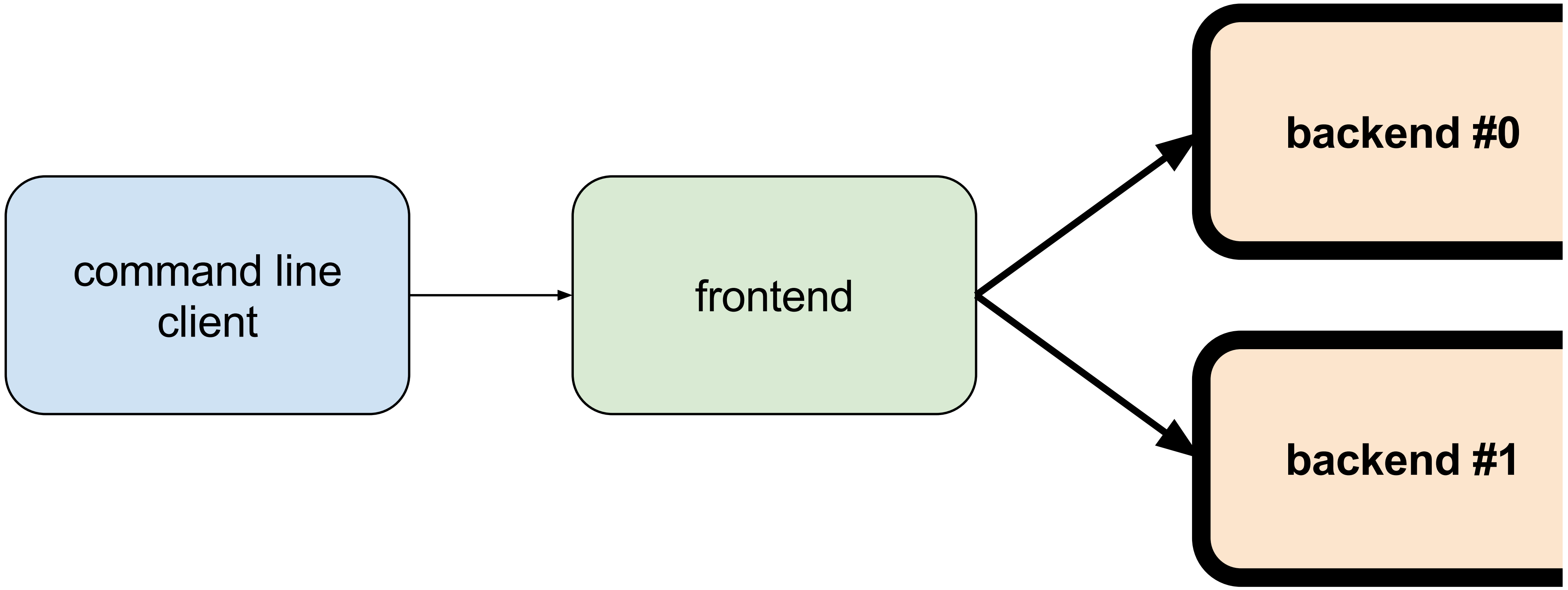
Client code (search)

```
func search(client pb.GoogleClient, query string) {
    ctx, cancel := context.WithTimeout(context.Background(), 80*time.Millisecond)
    defer cancel()
    req := &pb.Request{Query: query}
    res, err := client.Search(ctx, req)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(res)
}
```

RPCs block but can be canceled using a Context.

gRPC propagates cancelation from client to server.

Backend code



Backend code (main)

```
    lis, err := net.Listen("tcp", fmt.Sprintf(":%d", 36061+*index)) // RPC port
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    g := grpc.NewServer()
    pb.RegisterGoogleServer(g, new(server))
    g.Serve(lis)
```

new(`server`) must implement the `GoogleServer` interface:

```
type GoogleServer interface {
    // Search returns a Google search result for the query.
    Search(context.Context, *Request) (*Result, error)
    // Watch returns a stream of Google search results for the query.
    Watch(*Request, Google_WatchServer) error
}
```

Each call to `Search` or `Watch` runs in its own goroutine.

Backend code (search)

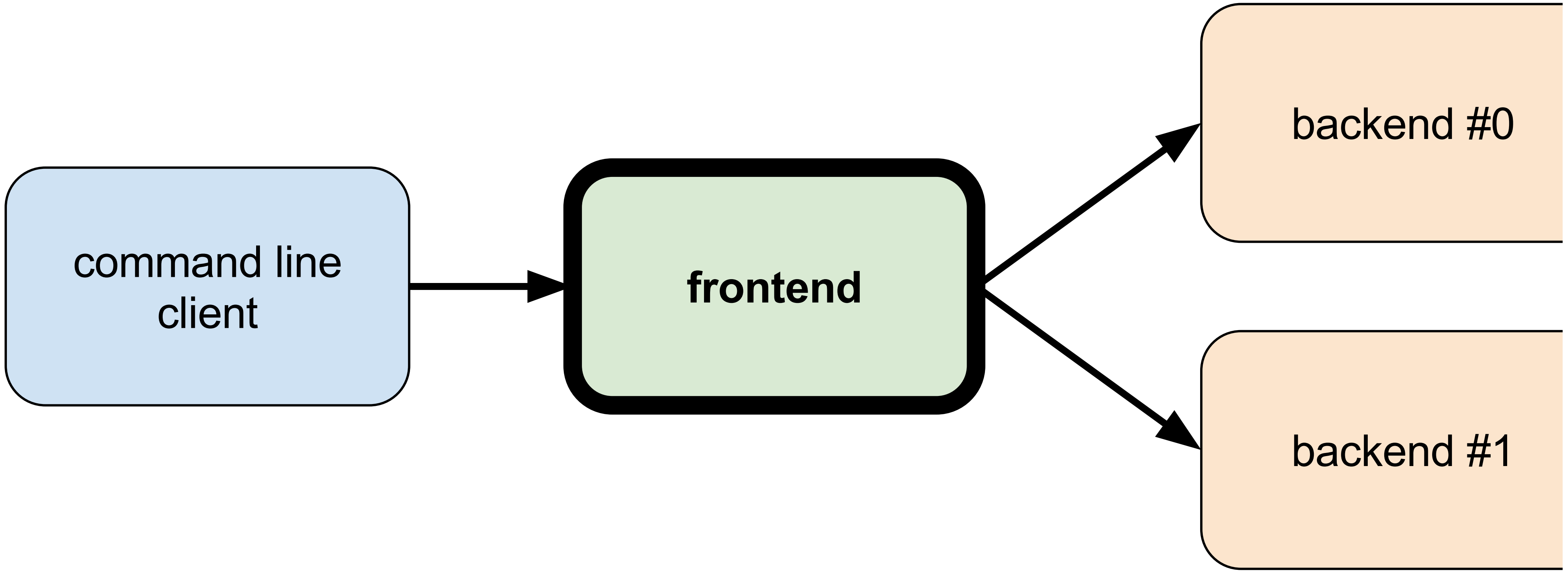
`ctx.`Done is closed when the RPC is canceled, times out, or returns:

```
func (s *server) Search(ctx context.Context, req *pb.Request) (*pb.Result, error) {
    d := randomDuration(100 * time.Millisecond)
    logSleep(ctx, d)
    select {
    case <-time.After(d):
        return &pb.Result{
            Title: fmt.Sprintf("result for [%s] from backend %d", req.Query, *index),
        }, nil
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}
```

If tracing is enabled, log the sleep duration:

```
func logSleep(ctx context.Context, d time.Duration) {
    if tr, ok := trace.FromContext(ctx); ok {
        tr.LazyPrintf("sleeping for %s", d)
    }
}
```

Frontend code



Frontend code (search)

`Search` returns as soon as it gets the first result.

gRPC cancels the remaining `backend.Search` RPCs by via `ctx`:

```go
func (s *server) Search(ctx context.Context, req *pb.Request) (*pb.Result, error) {
    c := make(chan result, len(s.backends))
    for _, b := range s.backends {
        go func(backend pb.GoogleClient) {
            res, err := backend.Search(ctx, req)
            c <- result{res, err}
        }(b)
    }
    first := <-c
    return first.res, first.err
}
```

```go
type result struct {
    res *pb.Result
    err error
}
```

Streaming RPC

Add Watch to the Google service

```
syntax = "proto3";

service Google {
  // Search returns a Google search result for the query.
  rpc Search(Request) returns (Result) {
  }
  // Watch returns a stream of Google search results for the query.
  rpc Watch(Request) returns (stream Result) {
  }
}

message Request {
  string query = 1;
}

message Result {
  string title = 1;
  string url = 2;
  string snippet = 3;
}
```

Generated code

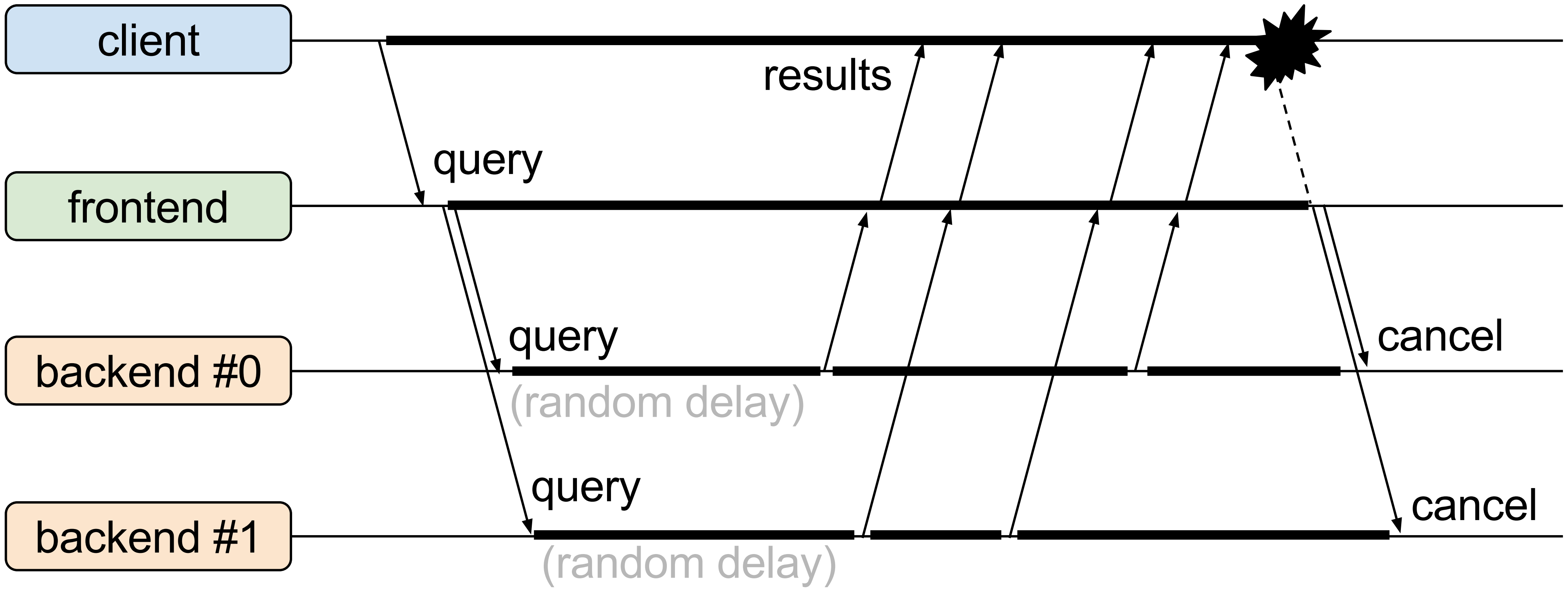
```
type GoogleClient interface {
    // Search returns a Google search result for the query.
    Search(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Result, error)
    // Watch returns a stream of Google search results for the query.
    Watch(ctx context.Context, in *Request, opts ...grpc.CallOption) (Google_WatchClient, error)
}
```

```
type GoogleServer interface {
    // Search returns a Google search result for the query.
    Search(context.Context, *Request) (*Result, error)
    // Watch returns a stream of Google search results for the query.
    Watch(*Request, Google_WatchServer) error
}
```

```
type Google_WatchClient interface {
    Recv() (*Result, error)
    grpc.ClientStream
}
```

```
type Google_WatchServer interface {
    Send(*Result) error
    grpc.ServerStream
}
```

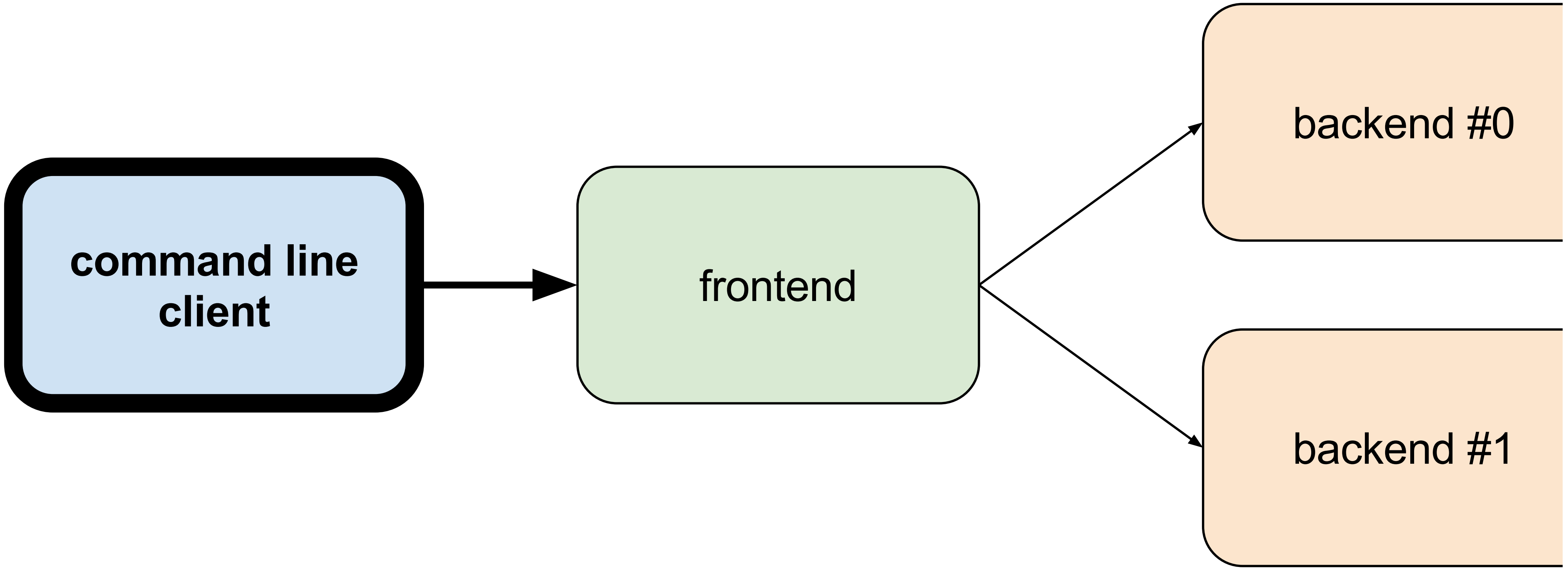
Frontend runs Watch on both backends and merges results



Demo client --mode=watch

- Active stream traces
- Cancelation

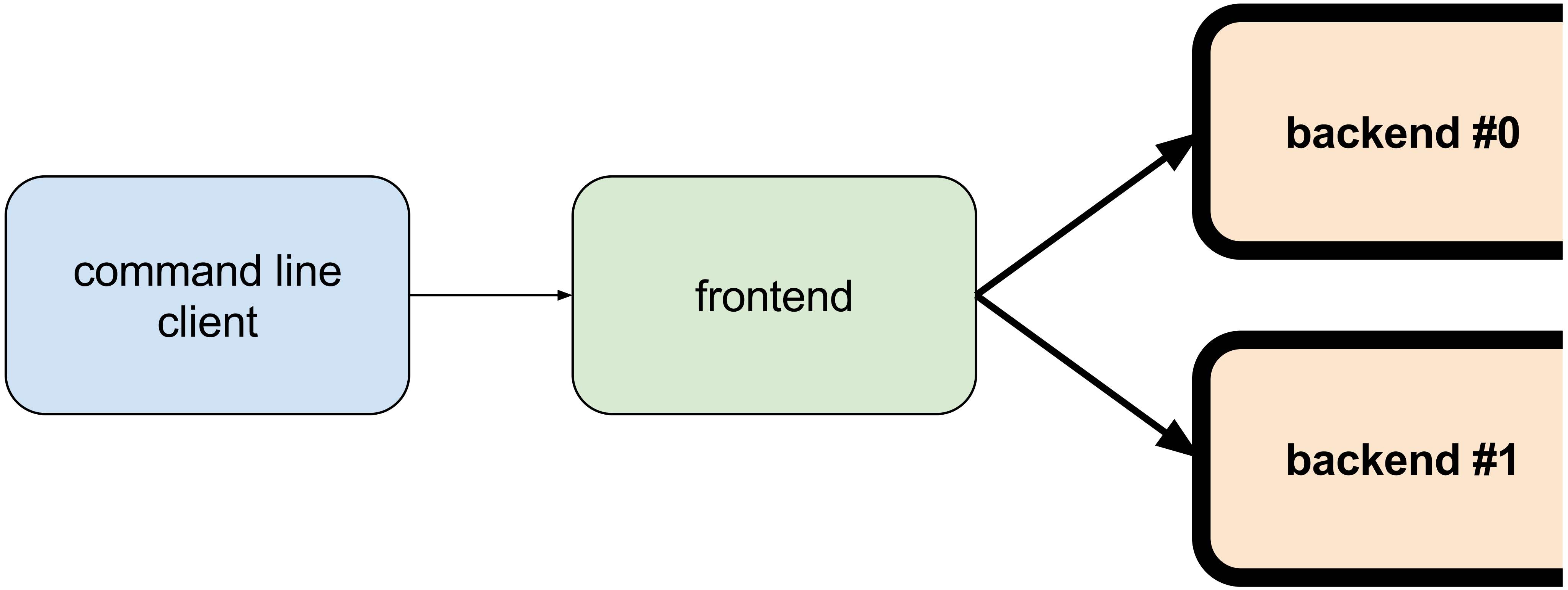
Client code



Client code (watch)

```go
func watch(client pb.GoogleClient, query string) {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    req := &pb.Request{Query: query}
    stream, err := client.Watch(ctx, req)
    if err != nil {
        log.Fatal(err)
    }
    for {
        res, err := stream.Recv()
        if err == io.EOF {
            fmt.Println("and now your watch is ended")
            return
        }
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println(res)
    }
}
```

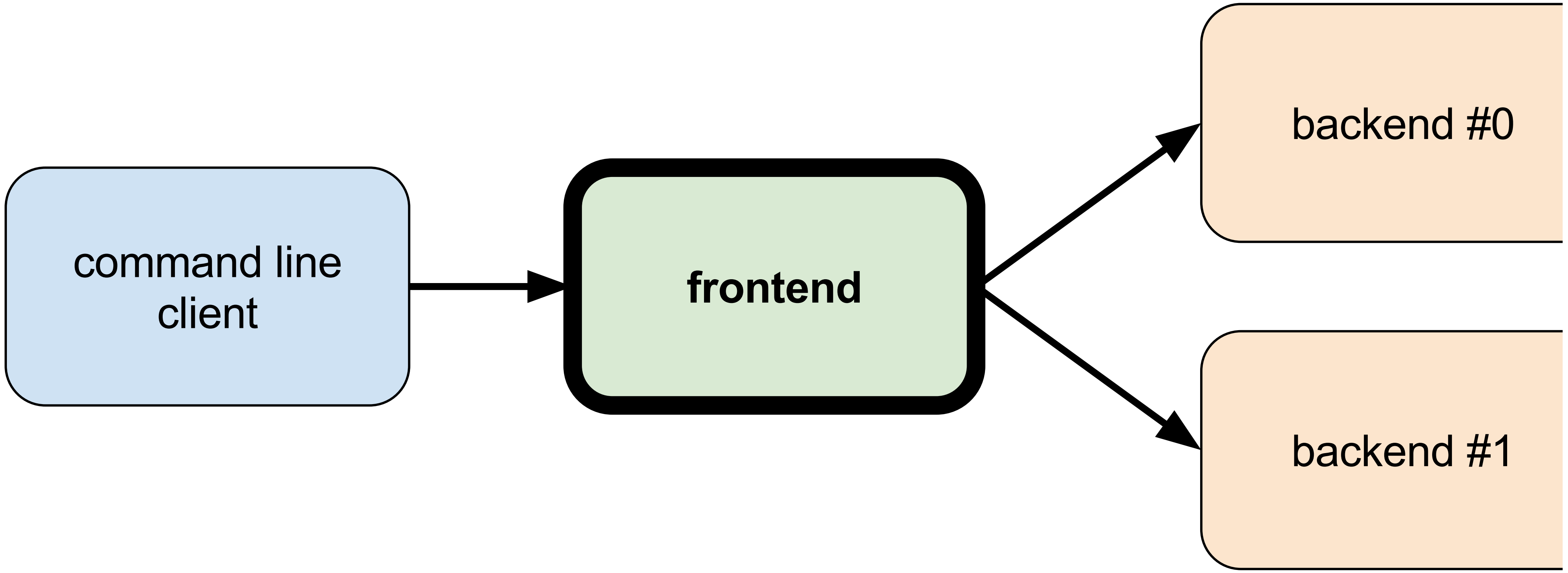
Backend code



Backend code (watch)

```
func (s *server) Watch(req *pb.Request, stream pb.Google_WatchServer) error {
    ctx := stream.Context()
    for i := 0; ; i++ {
        d := randomDuration(1 * time.Second)
        logSleep(ctx, d)
        select {
        case <-time.After(d):
            err := stream.Send(&pb.Result{
                Title: fmt.Sprintf("result %d for [%s] from backend %d", i, req.Query, *index),
            })
            if err != nil {
                return err
            }
        case <-ctx.Done():
            return ctx.Err()
        }
    }
}
```

Frontend code



Frontend code (watch)

```
func (s *server) Watch(req *pb.Request, stream pb.Google_WatchServer) error {
    ctx := stream.Context()
    c := make(chan result)
    var wg sync.WaitGroup
    for _, b := range s.backends {
        wg.Add(1)
        go func(backend pb.GoogleClient) {
            defer wg.Done()
            watchBackend(ctx, backend, req, c)
        }(b)
    }
    go func() {
        wg.Wait()
        close(c)
    }()
    for res := range c {
        if res.err != nil {
            return res.err
        }
        if err := stream.Send(res.res); err != nil {
            return err
        }
    }
    return nil
```

Frontend code (watchBackend)

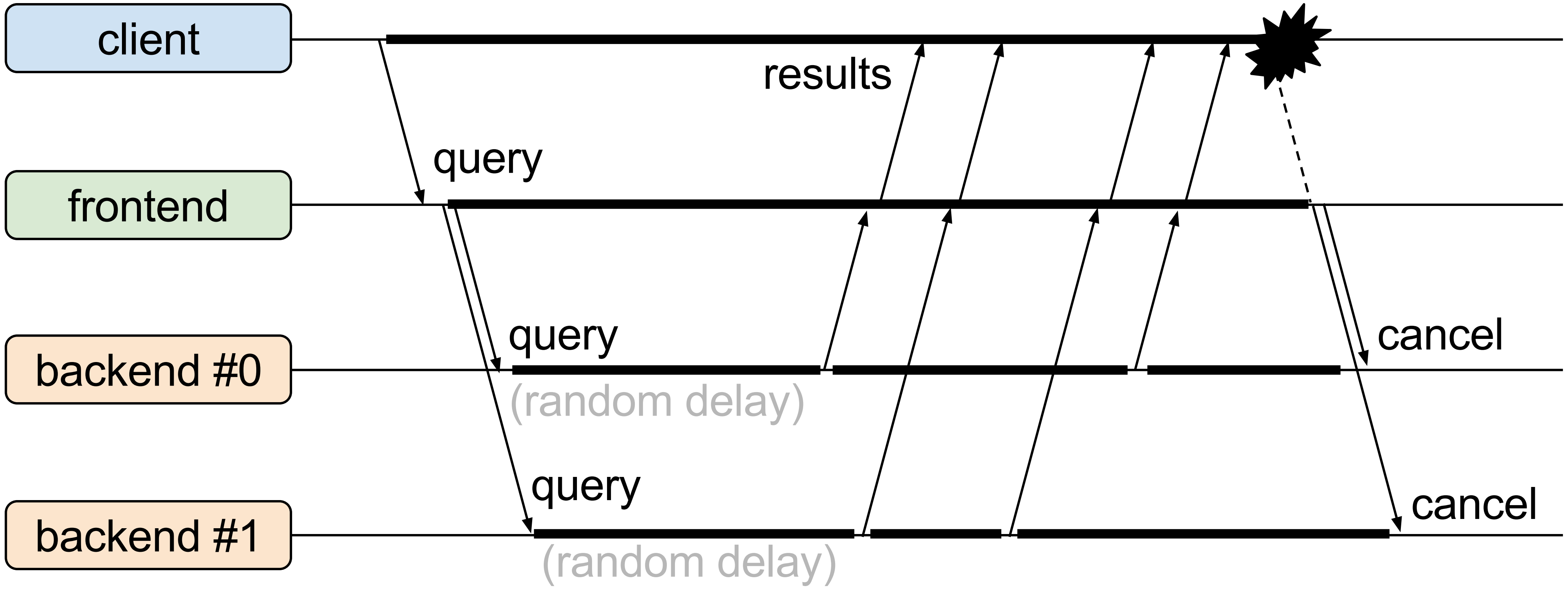
`Watch` returns on first error; this closes `ctx.Done` and signals `watchBackend` to exit.

```
func watchBackend(ctx context.Context, backend pb.GoogleClient, req *pb.Request, c chan<- result
) {
    stream, err := backend.Watch(ctx, req)
    if err != nil {
        select {
        case c <- result{err: err}:
        case <-ctx.Done():
        }
        return
    }
    for {
        res, err := stream.Recv()
        select {
        case c <- result{res, err}:
            if err != nil {
                return
            }
        case <-ctx.Done():
            return
        }
    }
}
```

gRPC extends the Go programming model over the network

Go gRPC works smoothly with goroutines, channels, and cancelation.

It is an excellent fit for building parallel, distributed, and streaming systems.



References

- grpc.io (http://grpc.io) - gRPC reference and tutorials
- github.com/golang/protobuf (https://github.com/golang/protobuf) - Protocol buffers
- golang.org/x/net/http2 (http://golang.org/x/net/http2) - HTTP2
- golang.org/x/net/trace (http://golang.org/x/net/trace) - Request traces and event logs
- golang.org/x/net/context (http://golang.org/x/net/context) - Cancelation and request-scoped data
- blog.golang.org/pipelines (http://blog.golang.org/pipelines) - Streaming data pipelines

Thanks to Qi Zhao, David Symonds, Brad Fitzpatrick, and the rest.

Questions?

Sameer Ajmani
Tech Lead Manager, Go team, Google
@Sajma (twitter.com/Sajma)
sameer@golang.org (mailto:sameer@golang.org)

Thank you

Sameer Ajmani
Tech Lead Manager, Go team, Google
@Sajma (http://twitter.com/Sajma)
sameer@golang.org (mailto:sameer@golang.org)