Distributed Systems
Lecture 10

Distributing a System

Joseph Phillips
Copyright (c) 2019
Last modified 2019 December 29

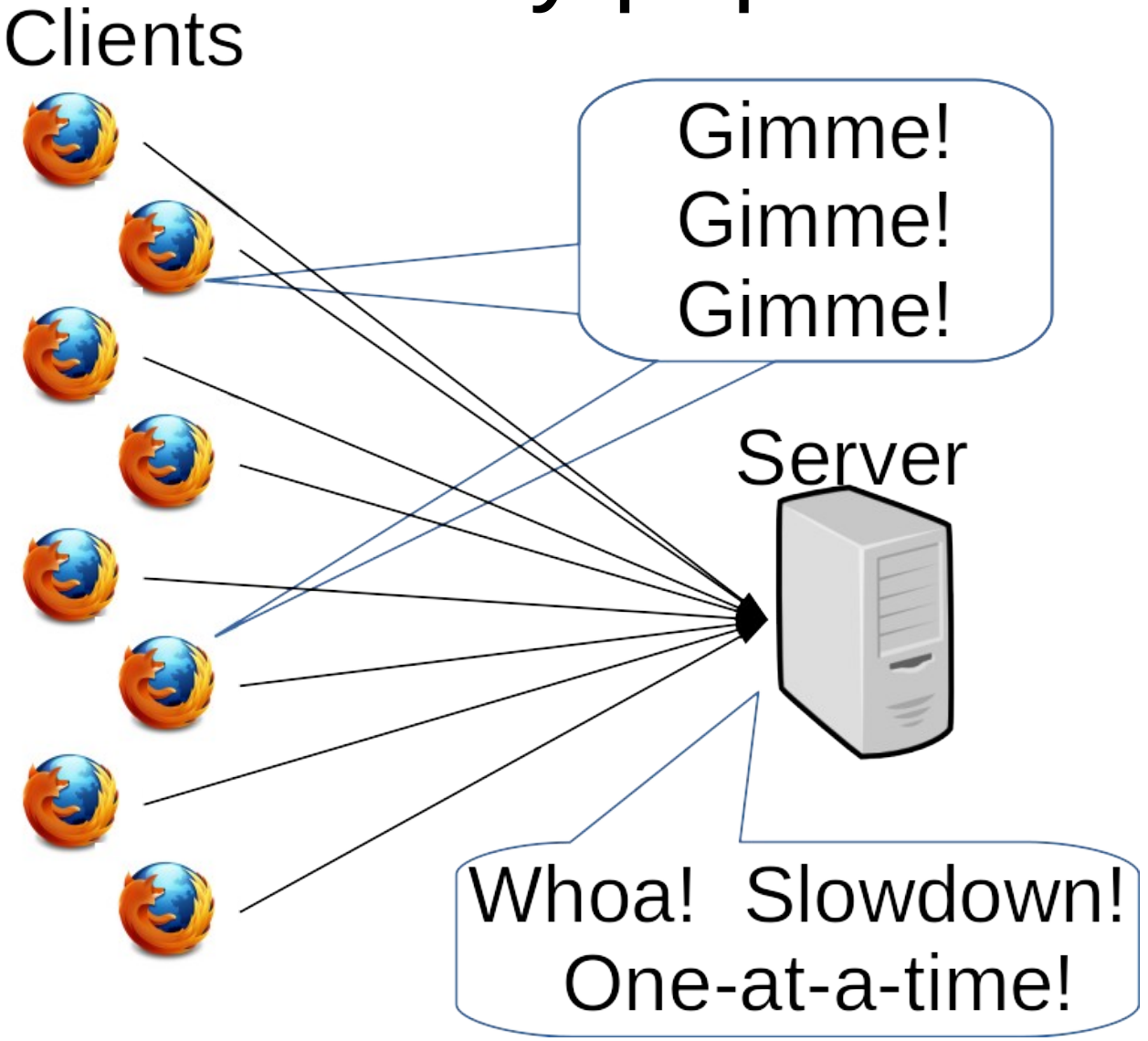
Reading

- "*Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*" by Brendan Burns, O'Reilly

Topics

- Distributing a System

We have a system!
It is very popular!

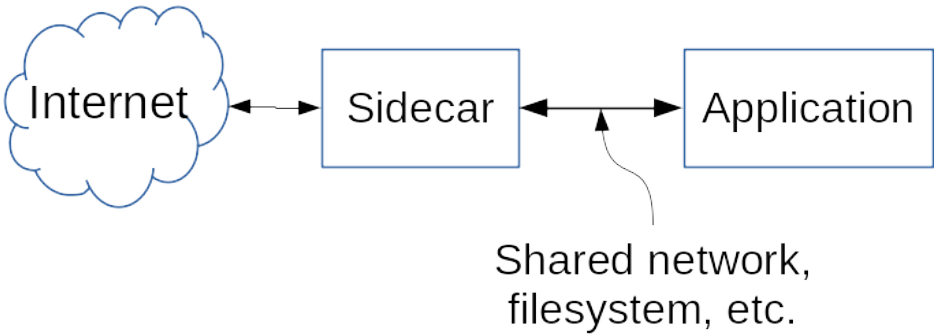


Time to scale up!

If only we knew how?

Hey! Let us consider design-patterns for distributed systems!

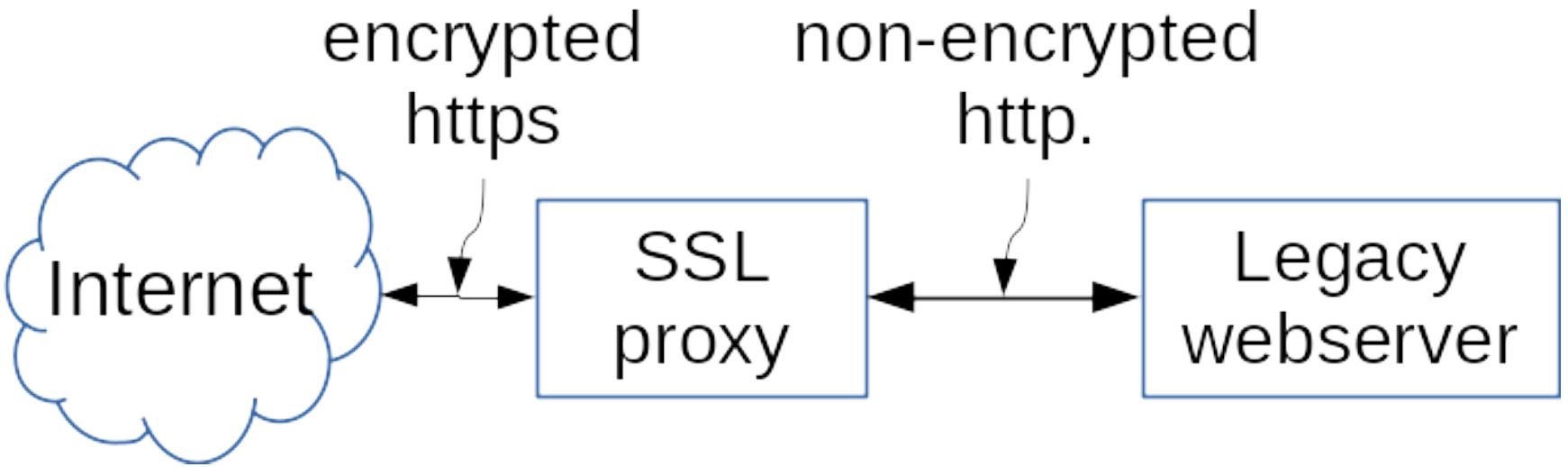
Sidecar Design Pattern



- Goal: Improve functionality of application (without app knowing it)

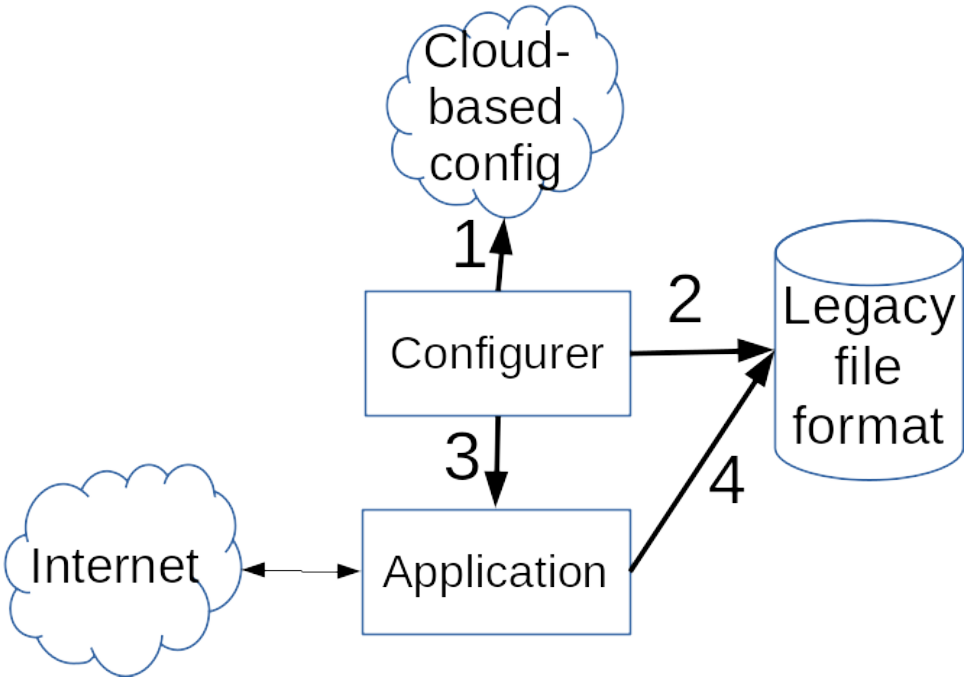
Side-car in use

- Example: Add SSL front end to legacy server



Side-car in use

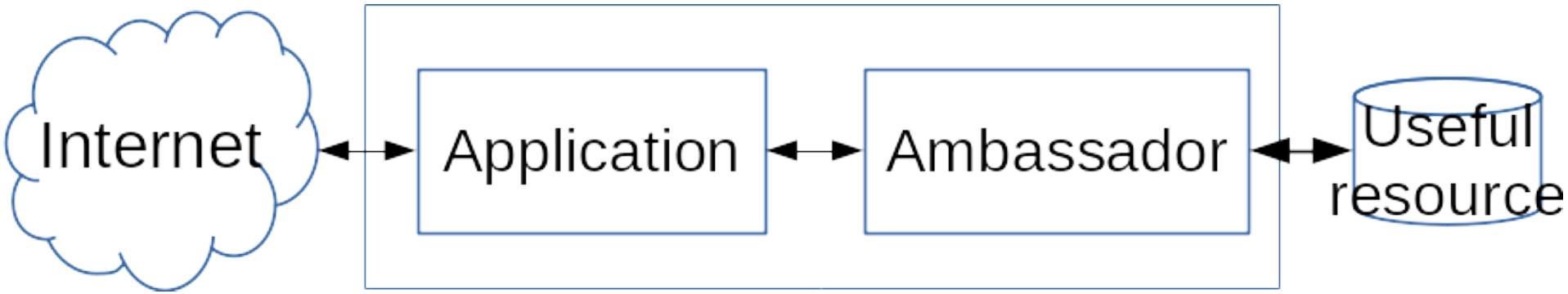
- Example: Allow dynamic configuration of legacy application



1) Configuration process continually checks cloud-based config files
2) When change noted, it writes to legacy file in legacy format
3) Config process then notifies (e.g. signals or restarts) application
4) App reloads legacy file

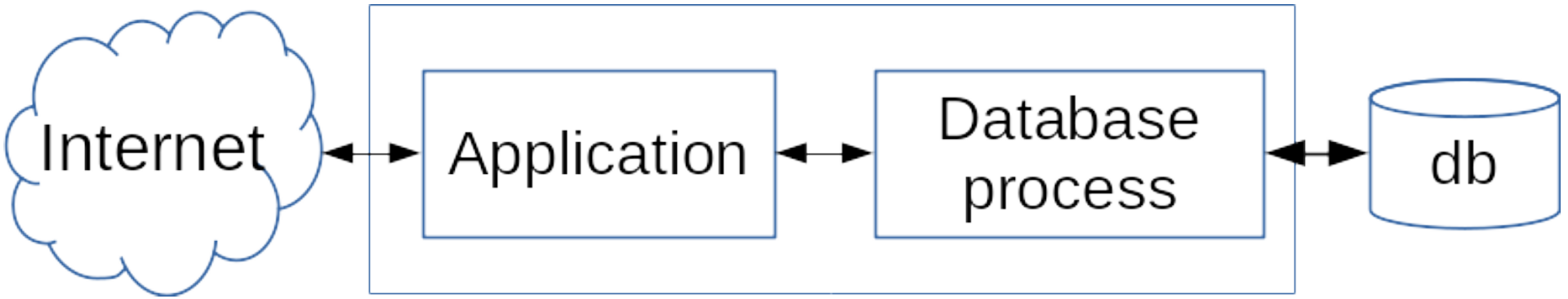
Ambassador Design Pattern

- Goal: To broker between application and other resources it might need



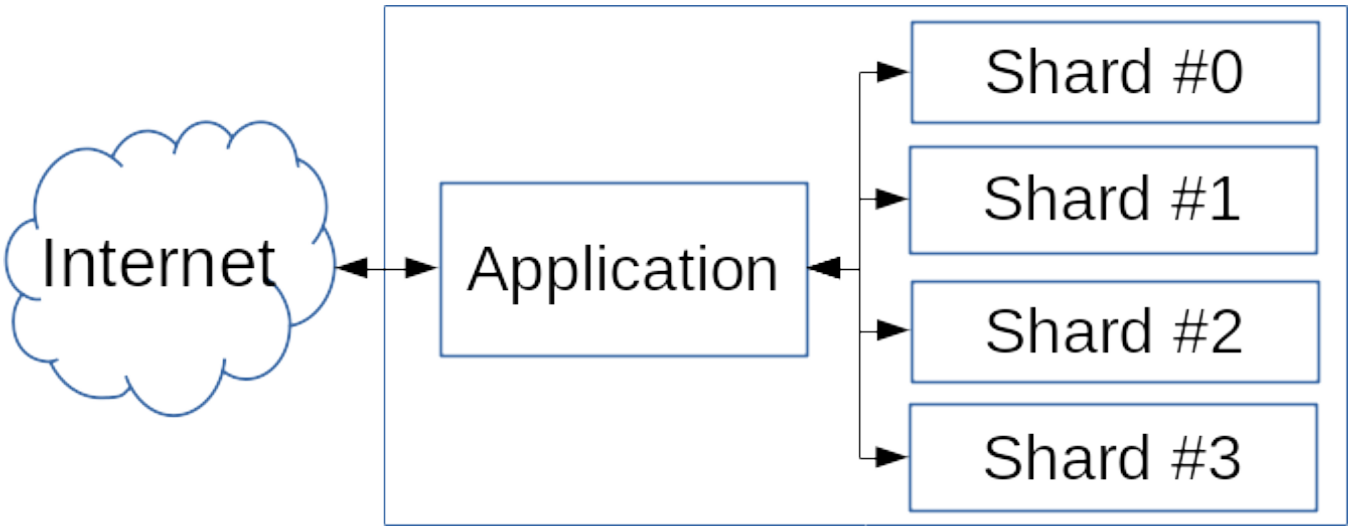
Ambassador in use

- Example: Database
  - Prototypical example of ambassador pattern



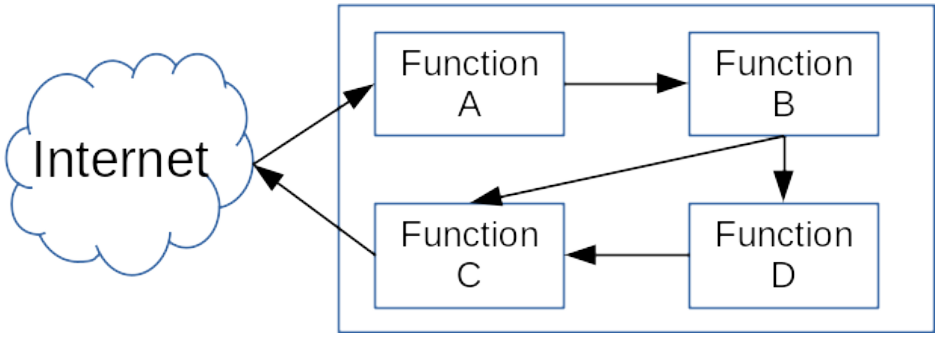
Ambassador in use

- Example: Multi-sharded application
  - Different shards have different responsibilities
  - E.g. handle different users



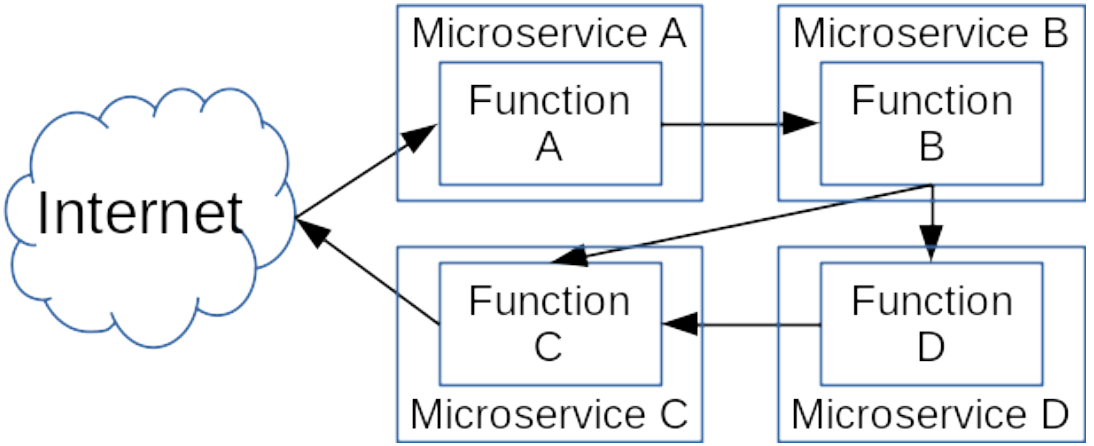
Serving Patterns

- We have 2 basic design patterns:
  - sidecar (app behind some helper process)
  - ambassador (app in front of some helper process)
- Let us apply them to larger, more realistic, systems
- ***But first***, how shall we even design a larger system?



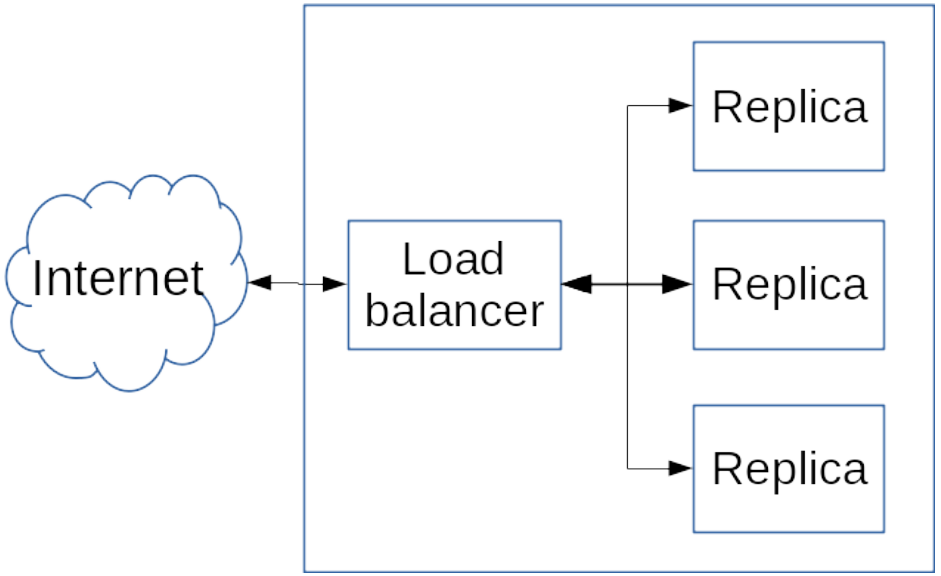
- The *Monolithic* design
  - Any problems?

Serving Patterns



- The *Microservice* design
- More reliable
  - If one thing crashes, does not bring down everything
- More agile
  - Need more of one particular service? ***No problem!***
  - Make more processes of just that desired microservice

Load balanced Architecture



- Load balancer is an application of ***sidecar***
- Which replica gets which request?
  - Need a hash-function
- What about IP-address?
  - Works okay for ***intra***net apps
  - Fails for ***Inter***net apps: IP addresses too clumped
- Consistent hash function
  - Allows adding/removing replicas as become more/less busy

Consistent hash function

- Motivation:
  - We dynamically add and remove servers
  - We do not want to drastically alter the hash function of clients to servers when we do so
  - (Thanks to Juan Pablo Carzollo, who attributes Karger et al from MIT)

InConsistent hash function

- serverToUse = userHash(user) % numServers
- 3 servers:
  - huey (0), dewey (1), louie (2)
- 4 users:
  - Alice (hash=50), Bob (hash=150), Cathy (hash=250), David (hash=350)
- Original mapping:
  - Alice: 50 % 3 = 2 (louie)
  - Bob: 150 % 3 = 0 (huey)
  - Cathy: 250 % 3 = 1 (dewey)
  - David: 350 % 3 = 2 (louie)

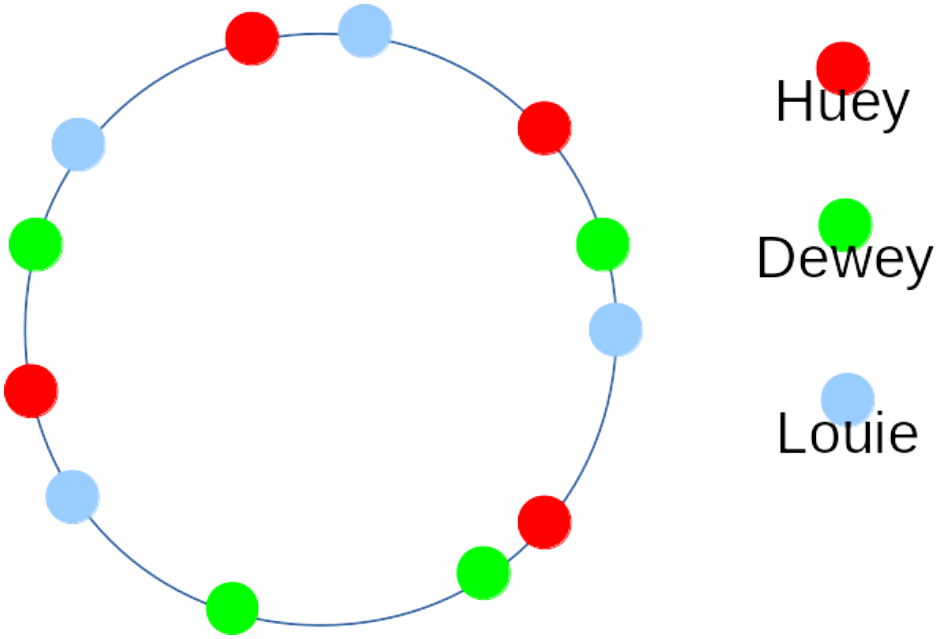
InConsistent hash function

- Now remove a server:
  - serverToUse = userHash(user) % numServers
- 2 servers:
  - huey (0), dewey (1)
- Many changes to the hash fnc!

- Original mapping:
  - Alice: 50 % 3 = 2 (louie)
  - Bob: 150 % 3 = 0 (huey)
  - Cathy: 250 % 3 = 1 (dewey)
  - David: 350 % 3 = 2 (louie)
- New mapping:
  - Alice: 50 % 2 = 0 (**huey**)
  - Bob: 150 % 2 = 0 (huey)
  - Cathy: 250 % 2 = 0 (**huey**)
  - David: 350 % 2 = 0 (**huey**)

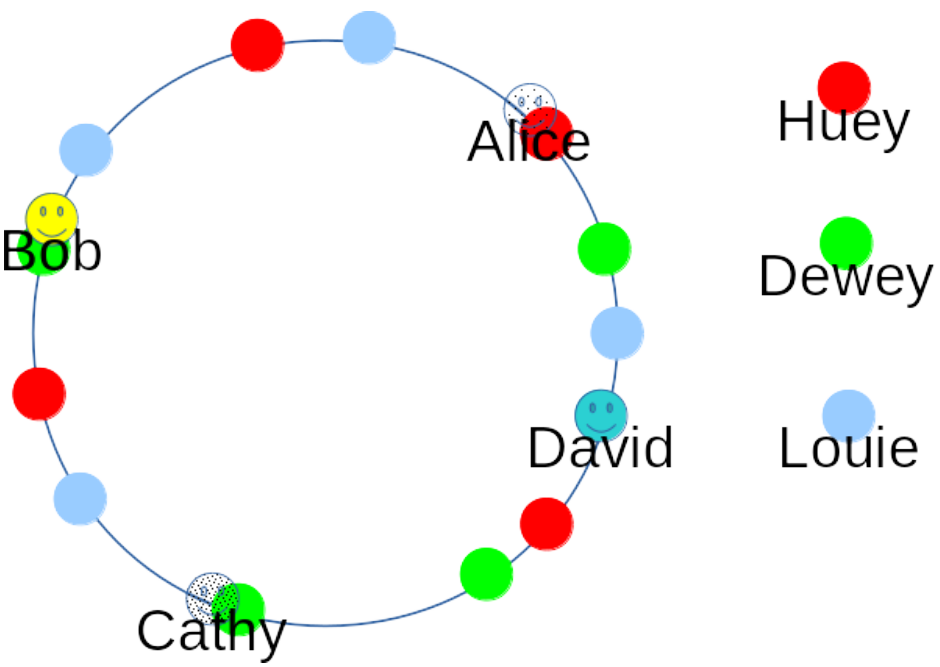
Consistent Hash Function (0)

- Consider a circle from 0 to 360 degrees
- Give each server several (in this case 4) random positions on the circle
- The 4 ***is this example only!*** In real life, you would have more!



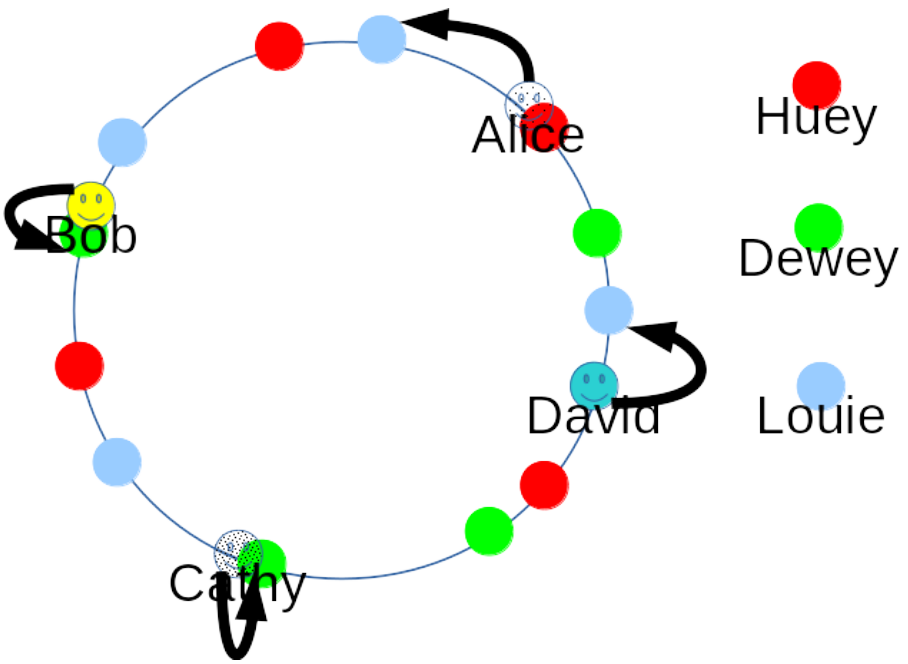
Consistent Hash Function (1)

- Now hash the users on the same circle:
  - userHash(user) % 360



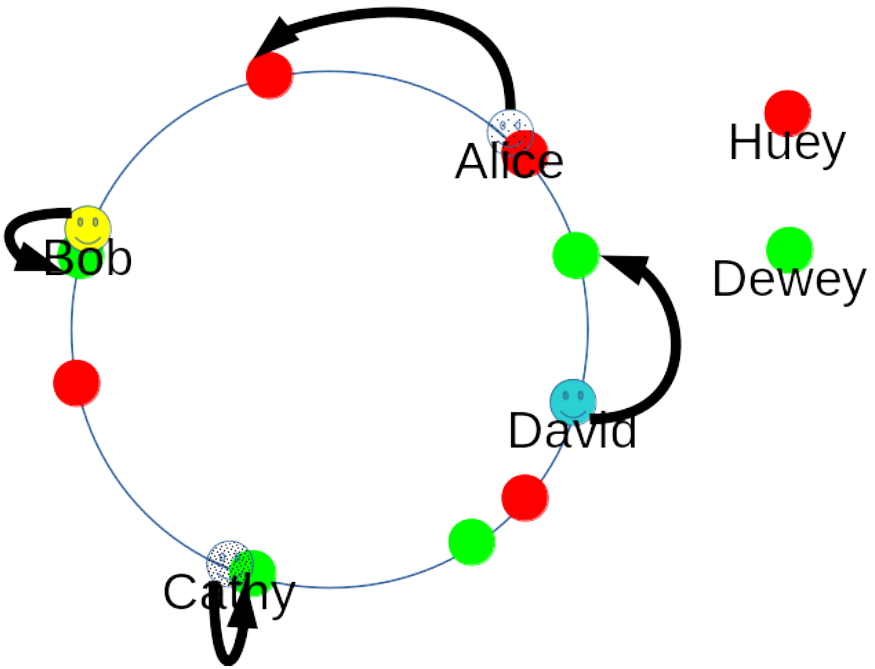
Consistent Hash Function (2)

- Map each user to a server by (for example) going counter-clockwise
  - Alice => Louie
  - Bob => Dewey
  - Cathy => Dewey
  - David => Louie



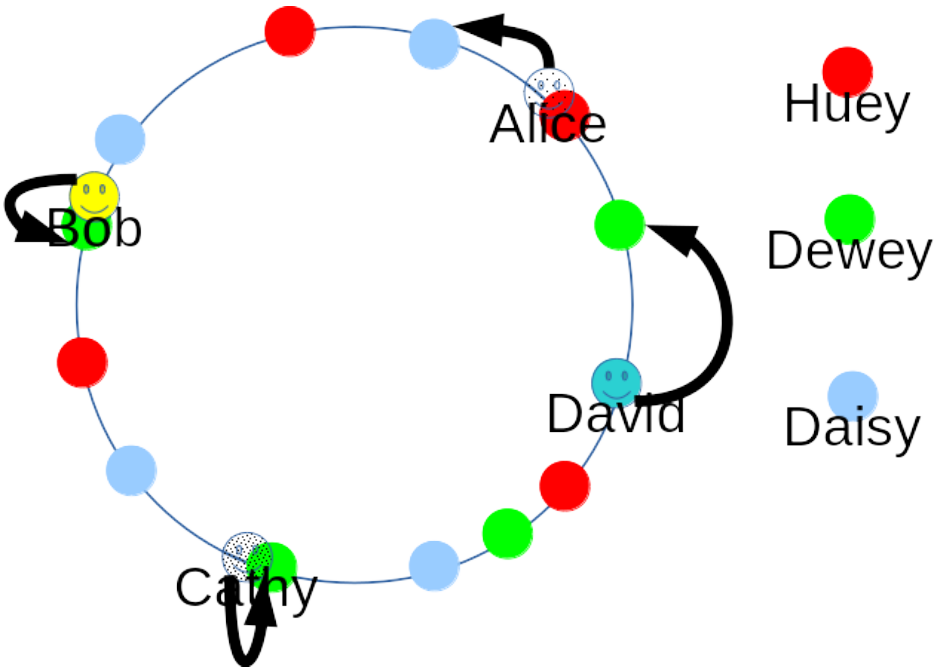
Consistent Hash Function (3)

- Remove a server!
- Only approx. ½ remaining mappings change:
  - Alice => **Huey**
  - Bob => Dewey
  - Cathy => Dewey
  - David => **Dewey**



Consistent Hash Function (4)

- Add a server!
- Only approx. 1/3 existing mappings change:
  - Alice => **Daisy**
  - Bob => Dewey
  - Cathy => Dewey
  - David => Dewey



Consistent Hash Function (5)

Q: “Hey, server Daisy is ***twice as powerful*** as Huey, Dewey or Louie!”

A: “Then give Daisy ***twice as many*** random points on the circle”

Consistent Hash Function (6)

- We will randomly spread points over a circle
- But what should ***userHash()*** be base on?
- Well, an HTTP request has several things including:
  - time (e.g. 12:00:01)
  - IP address (e.g. 1.2.3.4)
  - path (e.g. /directory/page.html)

Consistent Hash Function (7)

- So, let us consider them:
- Time
  - ***Too specific!*** 12:00:01 different from 12:00:02
- IP address
  - ***Too specific!*** 1.2.3.4 different from everything
- Path
  - ***Good idea!*** Go right to /directory/page.html

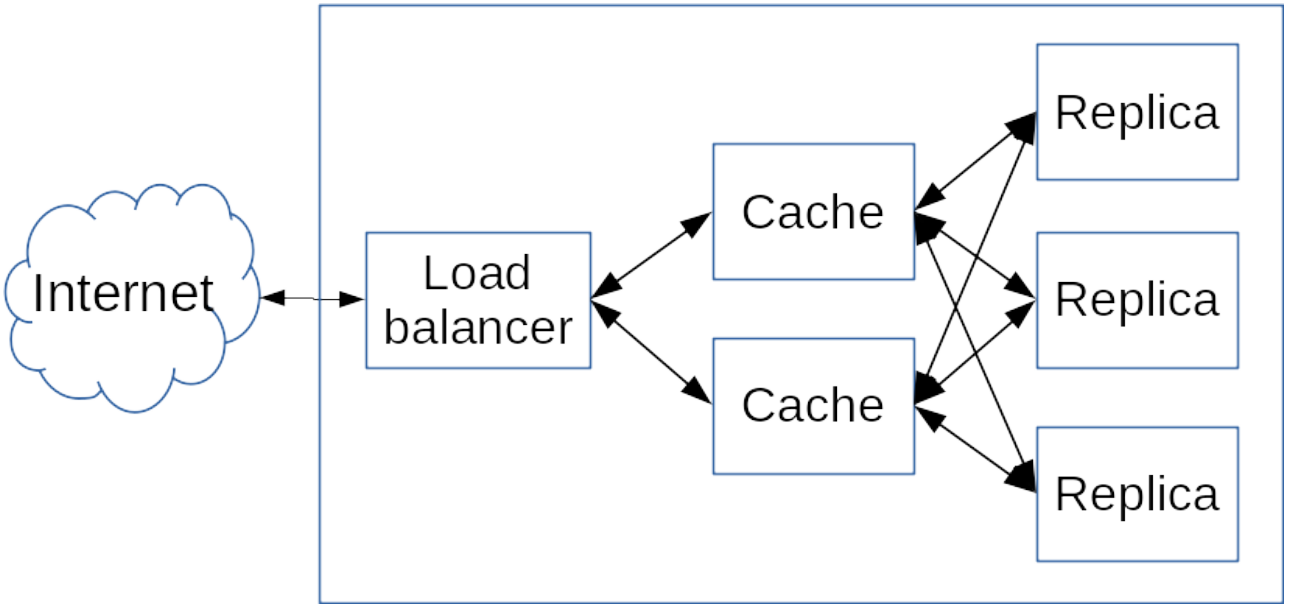
Consistent Hash Function (8)

- One teeny, tiny technicality . . . display these differently:
  - /directory/page.html (LANG=en_US)
  - /directory/page.html (LANG=es_MX)

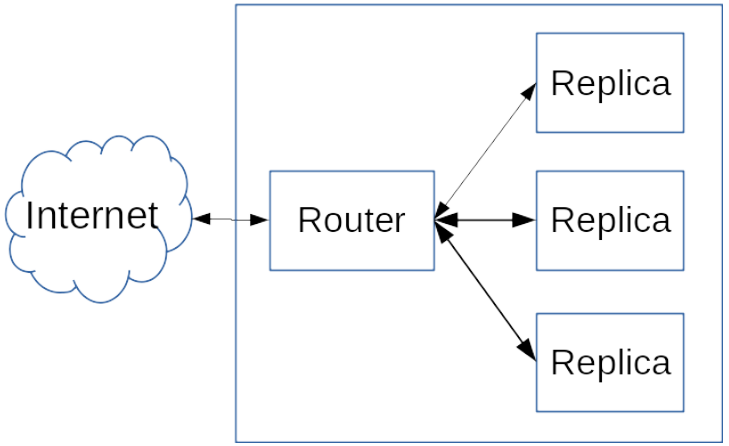
- What Brendan Burns recommends:
  - ***hash(country(request.ipAddr), request.path)***

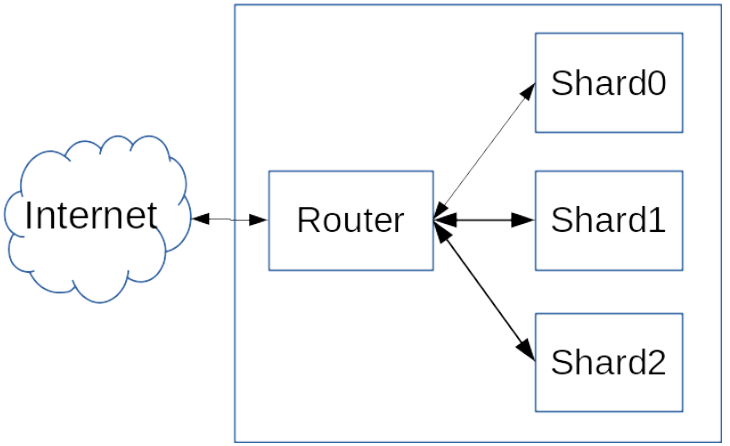
Load balanced Architecture

- Variations on a theme
  - Load balanced and cached architecture
- Look ma! ***Double sidecar!***
  - Cache's relationship with balancer
  - Replica's relationship with cache



Replicas vs. Shards





- Replicas:
  - All requests routed to all replicas
  - Gives ***redundancy***
- Shards:
  - Specific requests routed to specific shards
  - Gives ***efficiency***

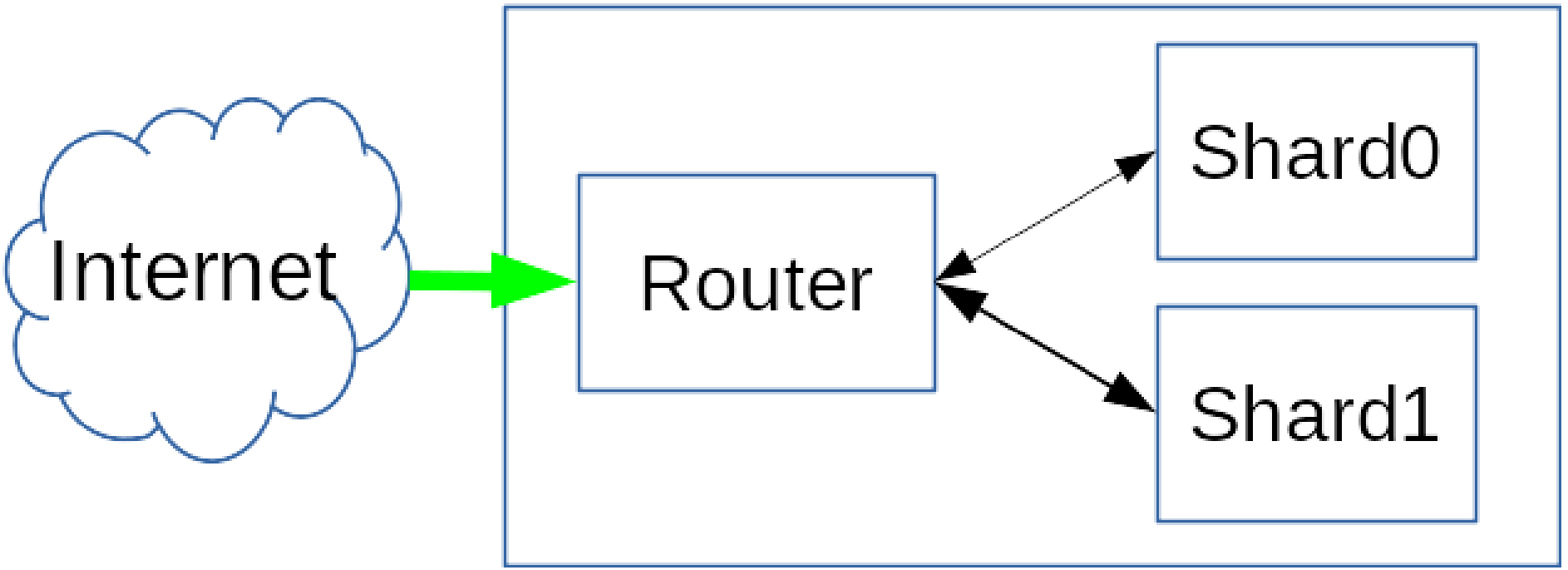
You can, of course, ***do both*** at different scales and levels!

Scatter/Gather Design Pattern

- Motivation:
  - Split requests up into multiple sub-requests
  - Give each sub-request to its own shard
- ***Look Ma!*** This is the ***ambassador design pattern*** applied multiple times

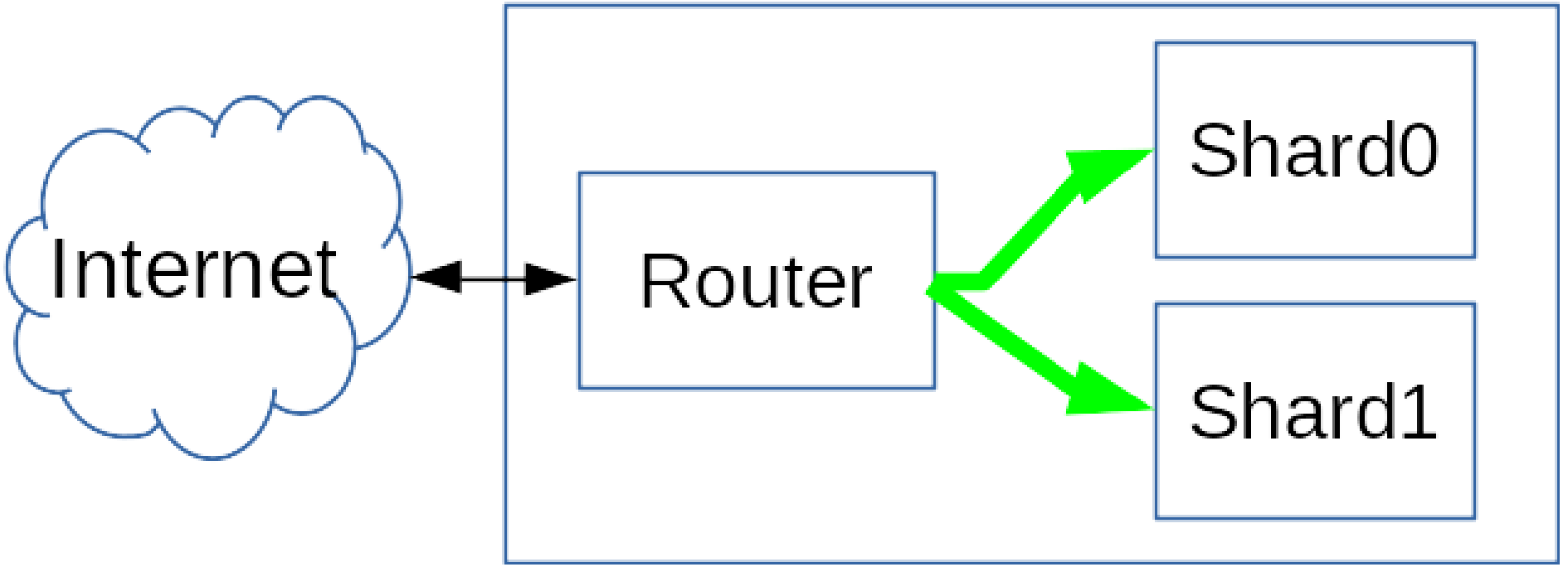
Scatter/Gather (0)

- Client "*Hey server: give me info about **chocolate** and **vanilla**!*"



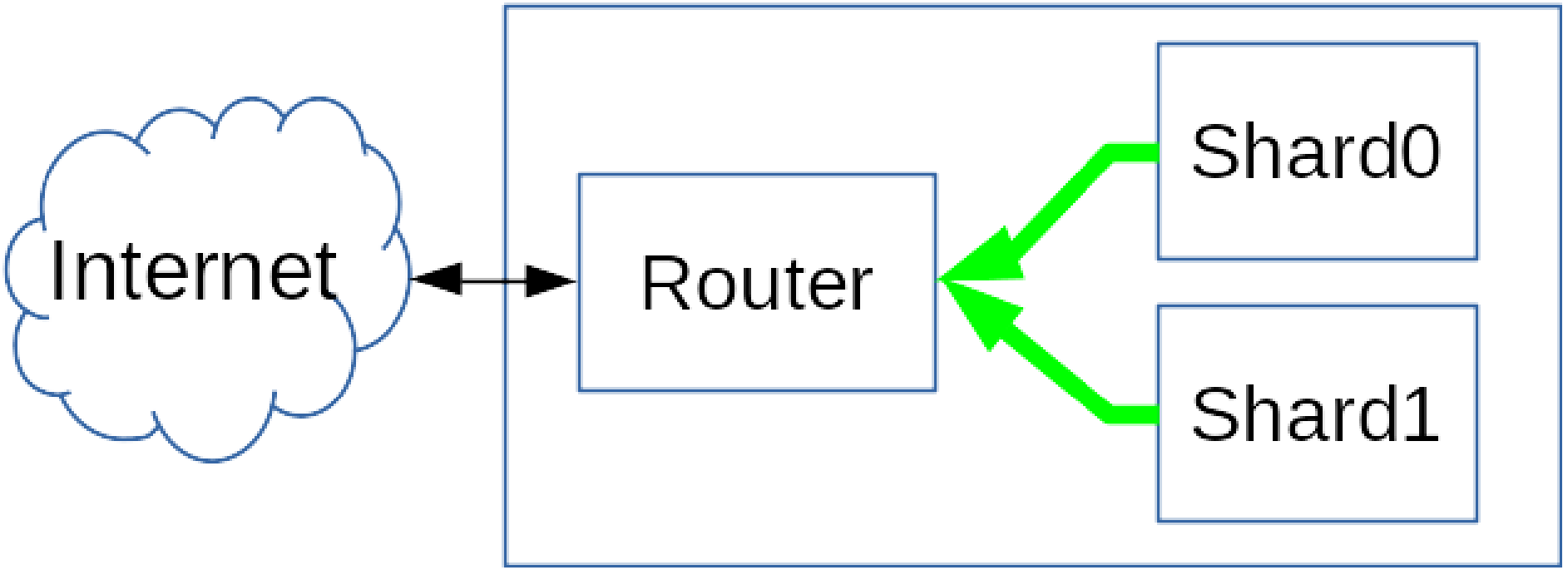
Scatter/Gather (1)

- Router/Combiner "*Shard0, you handle **chocolate**. Shard1, you handle **vanilla**.*"



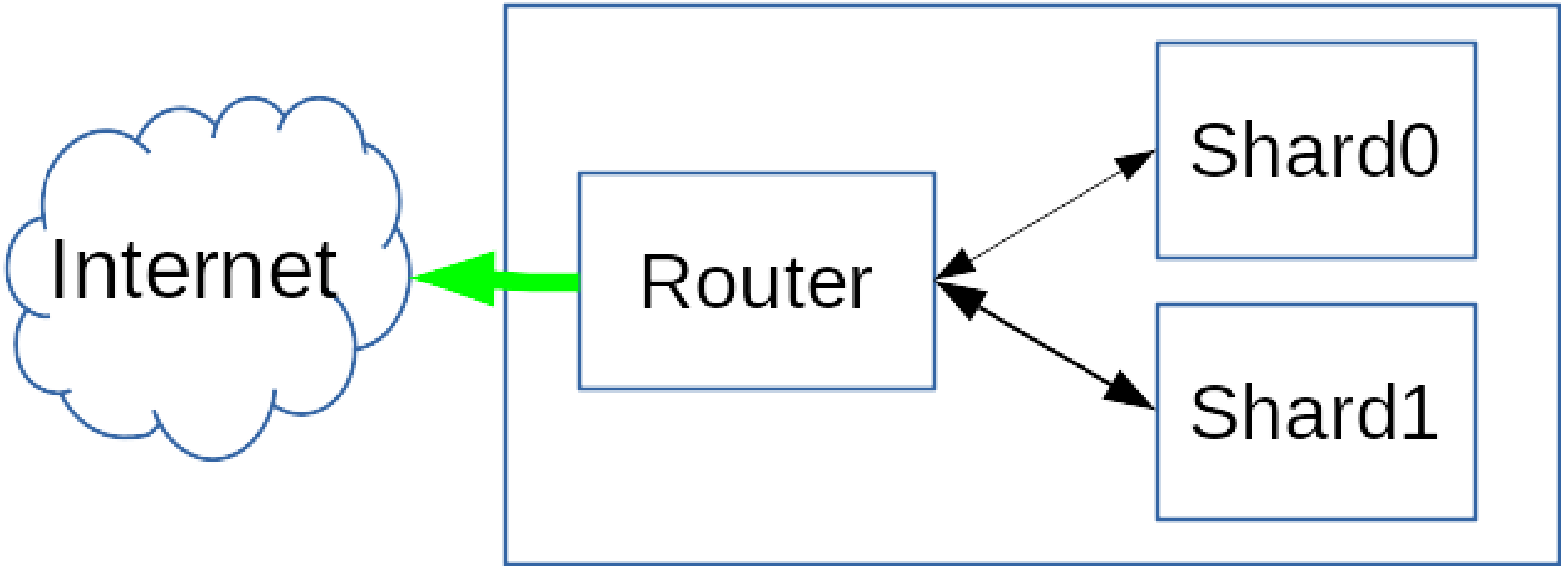
Scatter/Gather (2)

- Shard0 and Shard1 “*Here you go!*”



Scatter/Gather (3)

- Router/Combiner "*Chocolate and Vanilla, as you requested*"



References:

- Brendan Burns “*Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*”
- https://www.geeksforgeeks.org/parse-json-java/
- https://www.w3schools.com/js/js_json_xml.asp
- https://www.json.org/
- “The Ultimate Guide to Hashing Content” https://www.toptal.com/big-data/consistent-hashing Juan Pablo Carzollo, downloaded 2019 Nov 20
- Karger, David; Lehman, Eric; Leighton, Tom; Levine, Matthew; Lewin, Daniel; Panigrahy, Rina “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”