# Scalability

# Enterprise Scalability?

Web Scale!

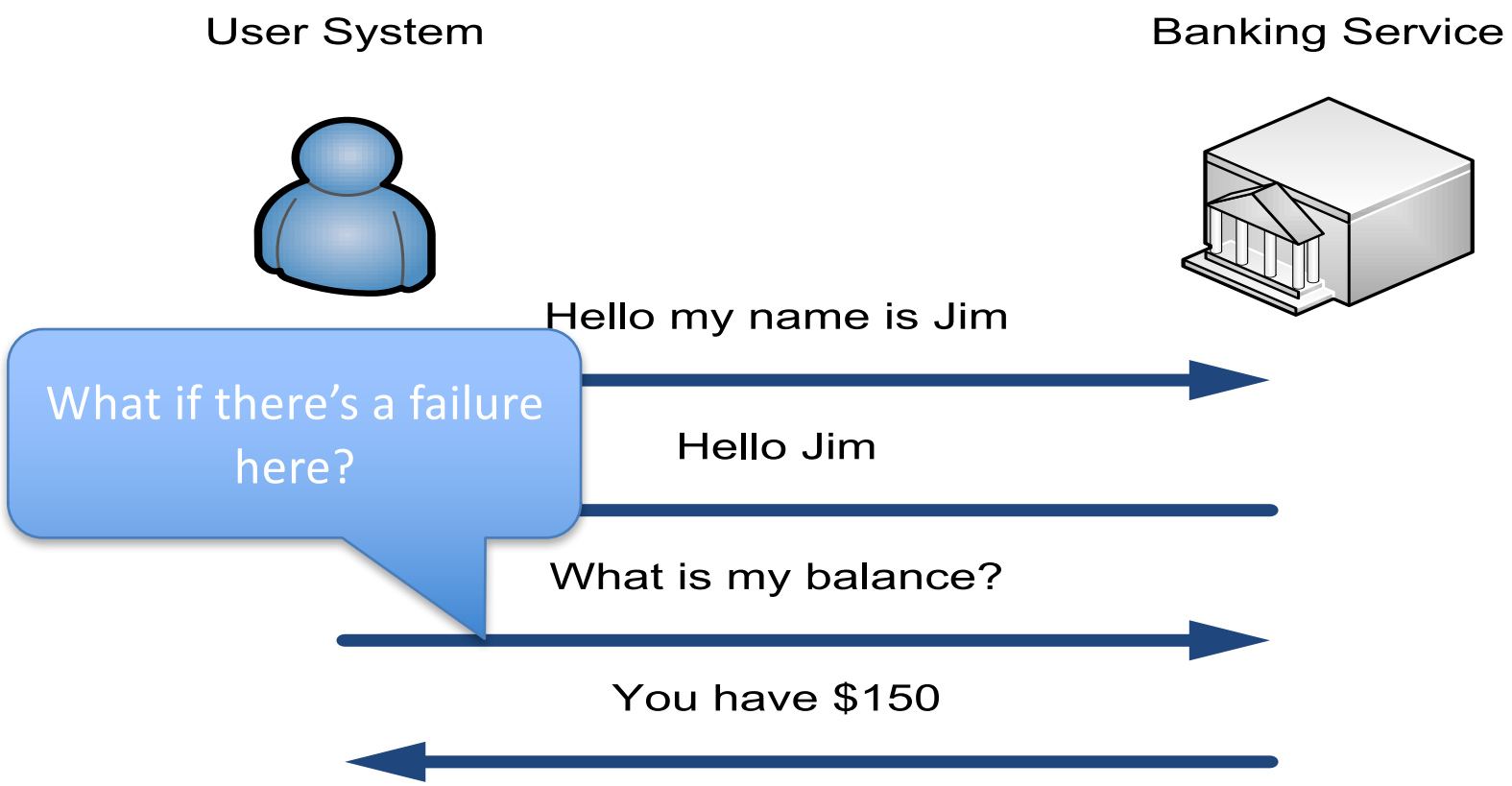
## Statelessness

- Every action happens in isolation
  - This is a good thing!
- In between requests the server knows nothing about you
  - Excepting any state changes you caused when you last interacted with it.
- Keeps the interaction protocol simpler
  - Makes recovery, scalability, failover much simpler too
  - Avoid cookies!

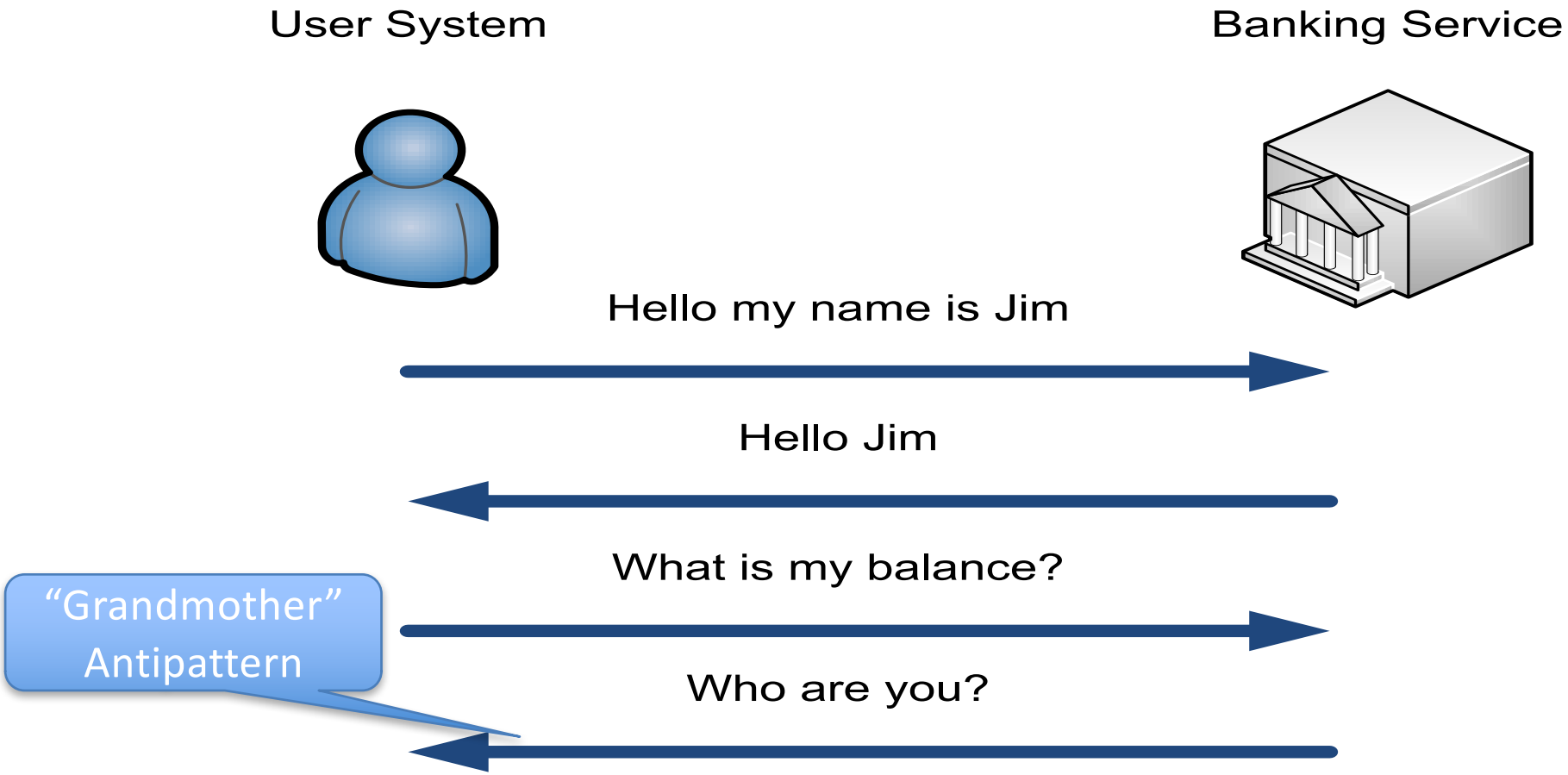
## Application vs Resource State

- Useful services hold persistent data – Resource state
  - Resources are buckets of state
  - What use is Google without state?
- Brittle implementations have application state
  - They support long-lived conversations
  - No failure isolation
  - Tricky crash recovery
  - Hard to scale, hard to do fail-over fault tolerance
- Recall stateless Web Services – same applies in the Web too!

# Stateful Example



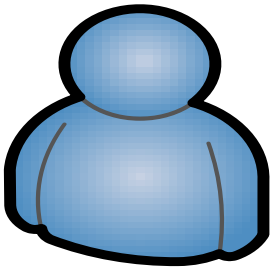
# Stateful Failure

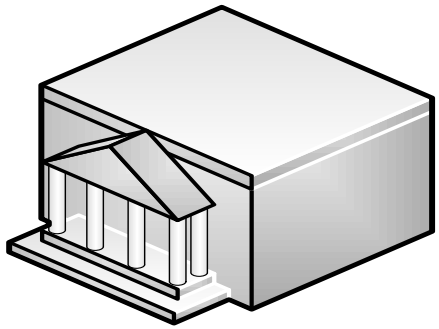


# Stateless System Tolerates Intermittent Failures

User System

Banking Service





I am Jim, what is my balance?

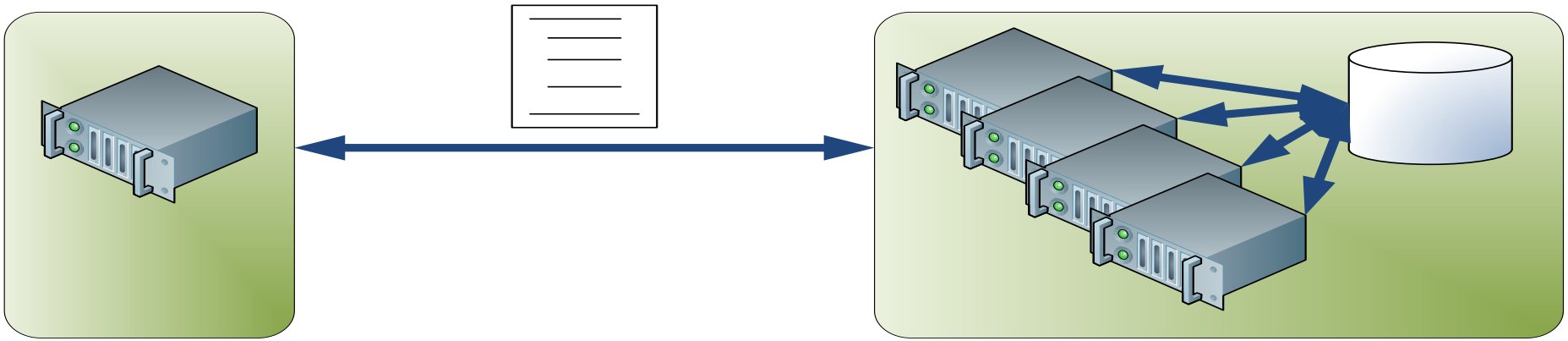
You have $150

## Scaling Horizontally

- Web farms have delivered horizontal scaling for years
  - Though they sometimes do clever things with session affinity to support cookie-based sessions
- In the programmatic Web, massive scalability enabled by
  - Stateless model
  - Caching
  - Work avoidance
- All are implicit parts of the application protocol (HTTP)

# Scalable Deployment Configuration

- Deploy services onto many servers
- Services are stateless
- Servers share only back-end data



## Scaling Vertically… without servers

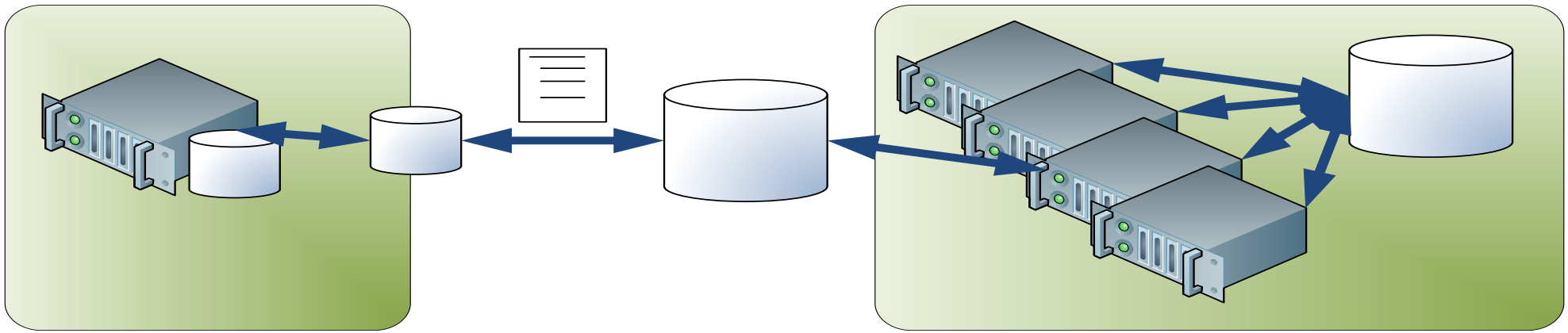
- The most expensive round-trip:
  - From client
  - Across network
  - Through servers
  - Across network again
  - To database
  - And all the way back!
- The Web tries to short-circuit this
  - By determining early if there is any actual work to do!
  - And by caching

## Caching

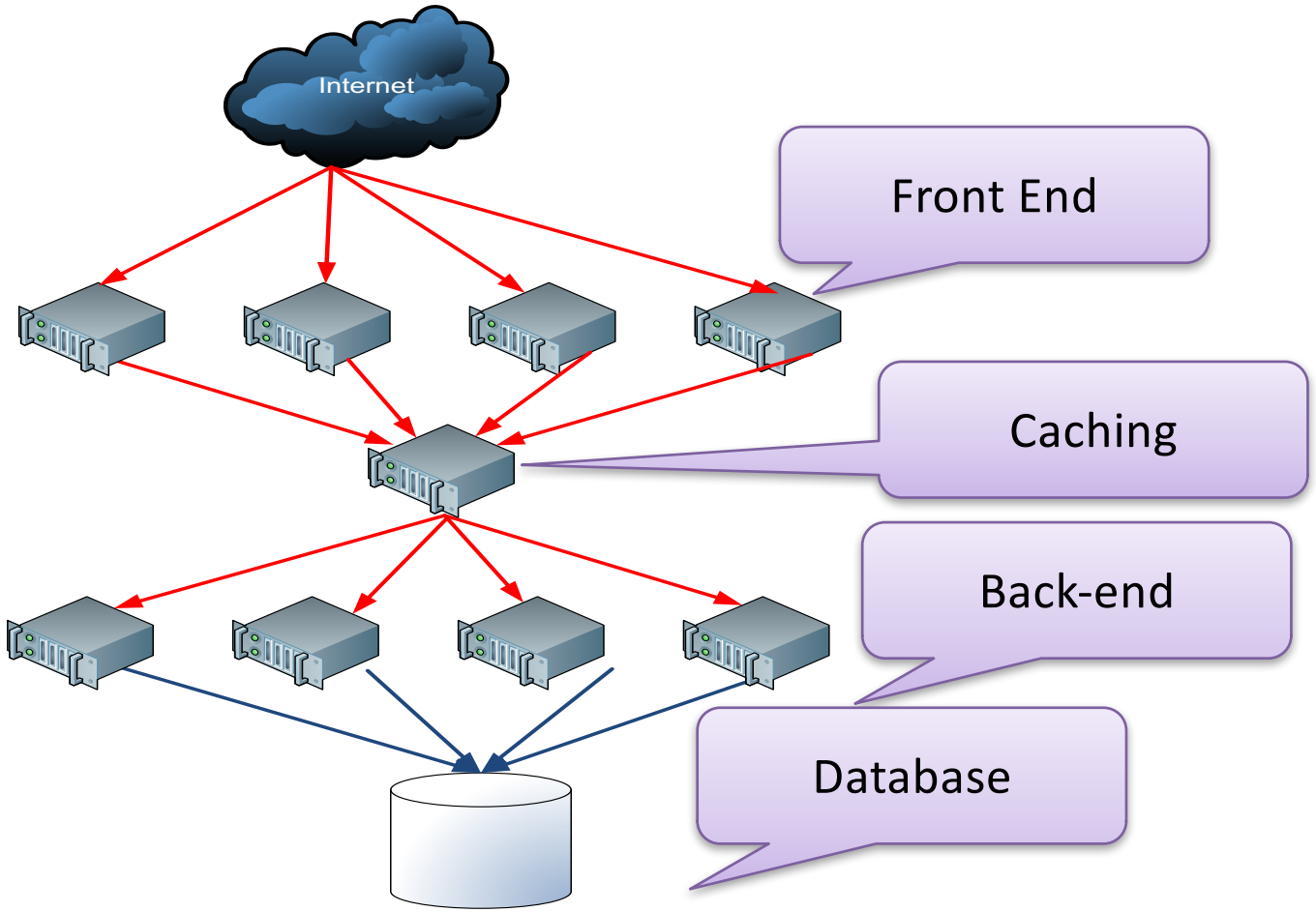
- Caching is about trading off freshness for throughput
  - Has side-effects for availability too
- Making a service run faster
  - Rather than getting higher overall throughput
- In the programmatic Web it's about reducing load on servers
  - And reducing *apparent* latency for clients

# Caching in a Scalable Deployment

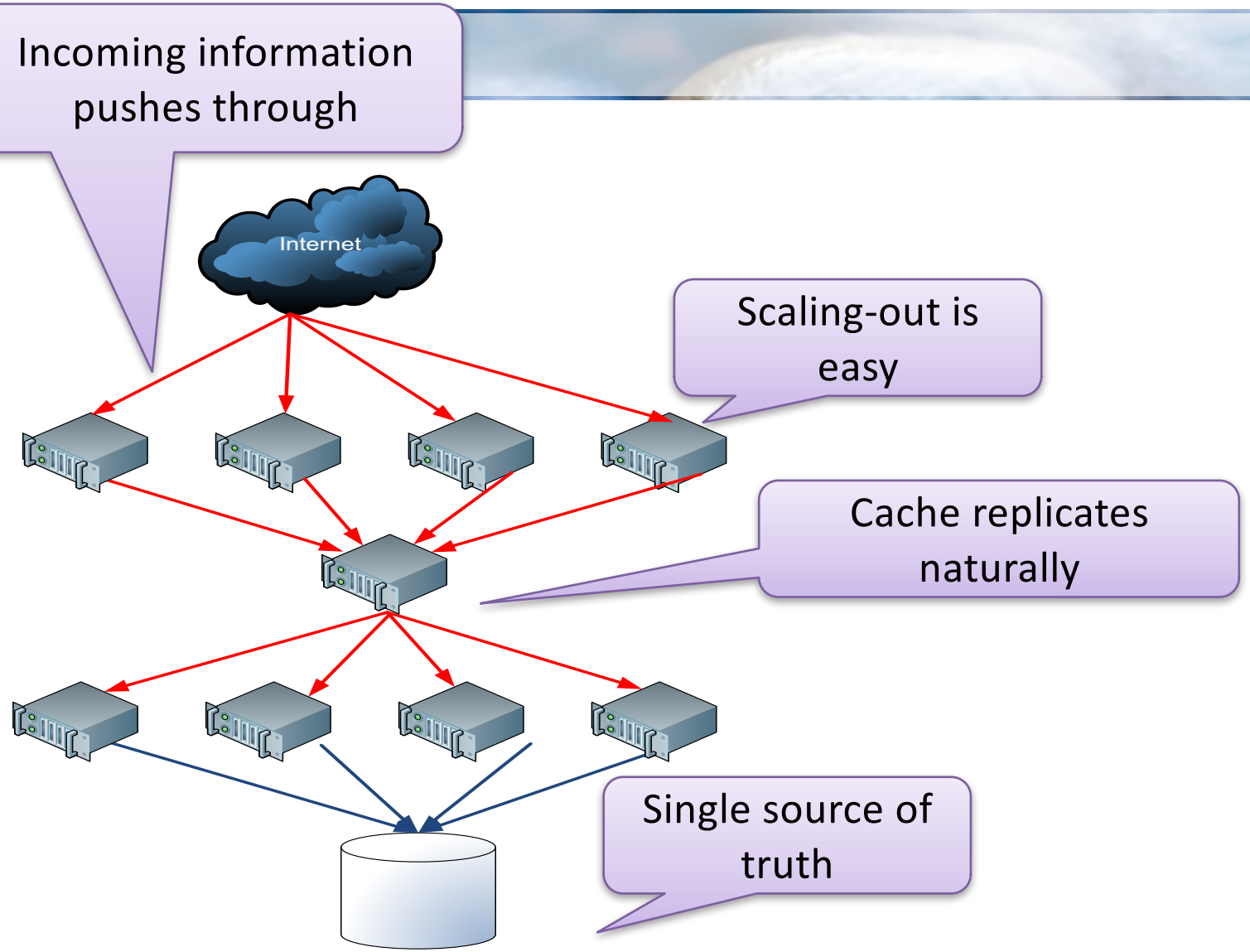
- Cache (reverse proxy) in front of server farm
  - Avoid hitting the server
- Proxy at client domain
  - Avoid leaving the LAN
- Local cache with client
  - Avoid using the network



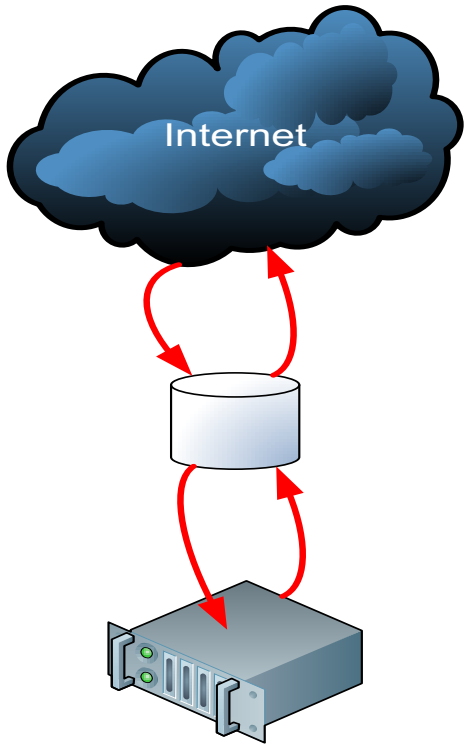
# Web-Inspired Architecture



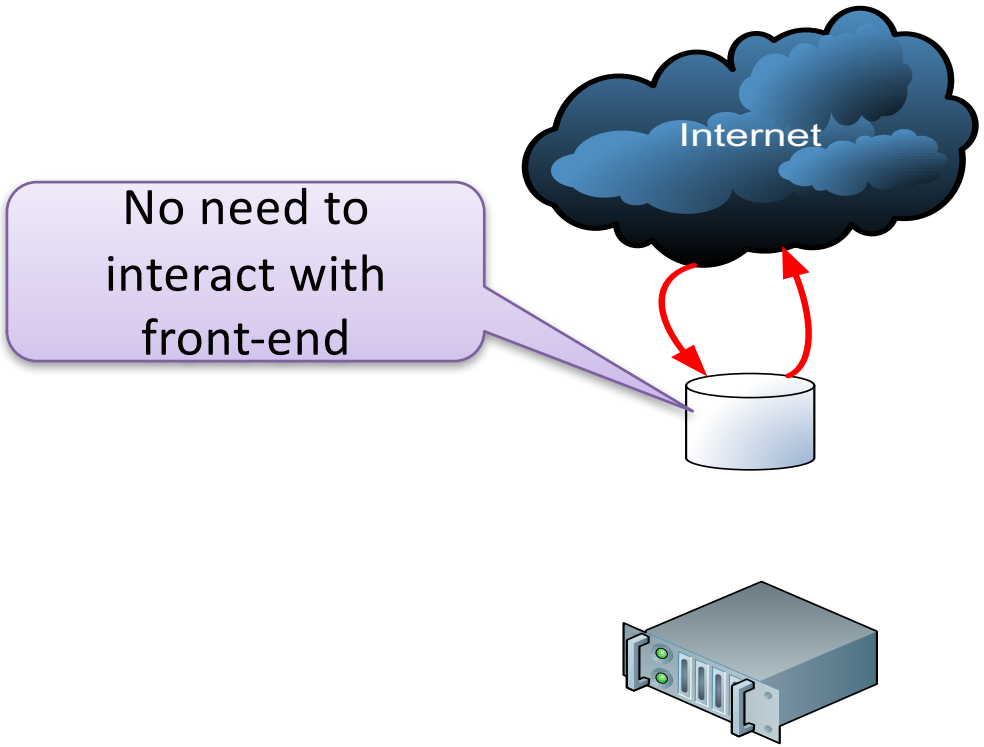
# Cha


Incoming information pushes through
Internet
Scaling-out is easy
Cache replicates naturally
Single source of truth

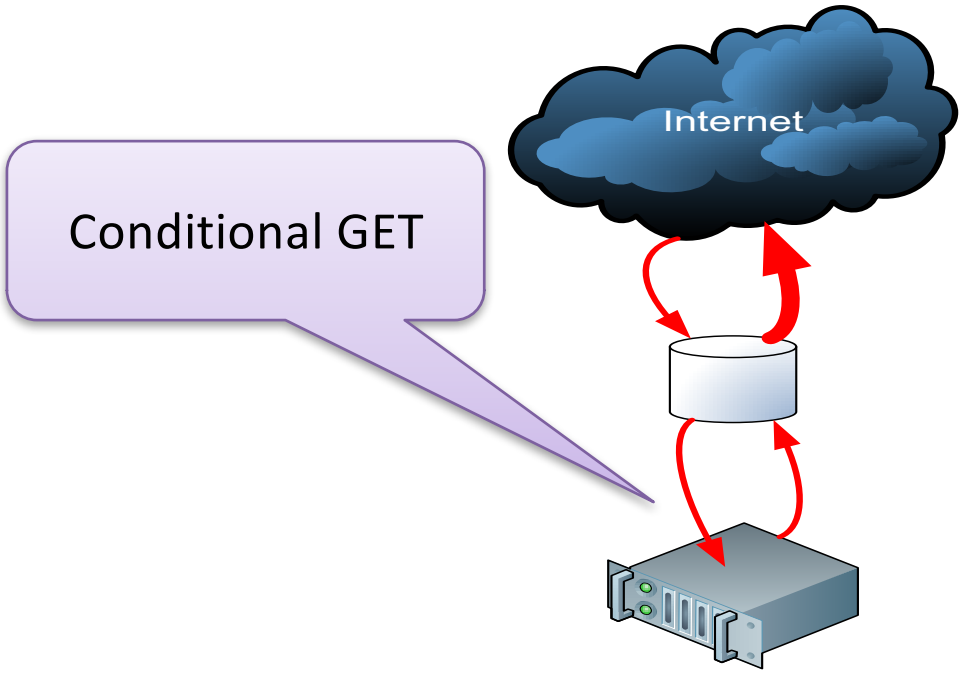
# Improving Performance with Caching



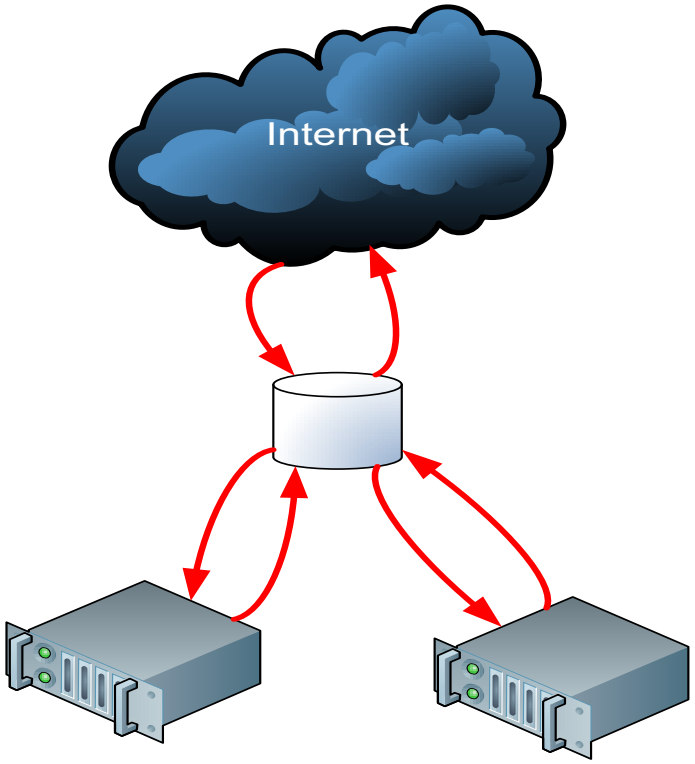
# Freshness



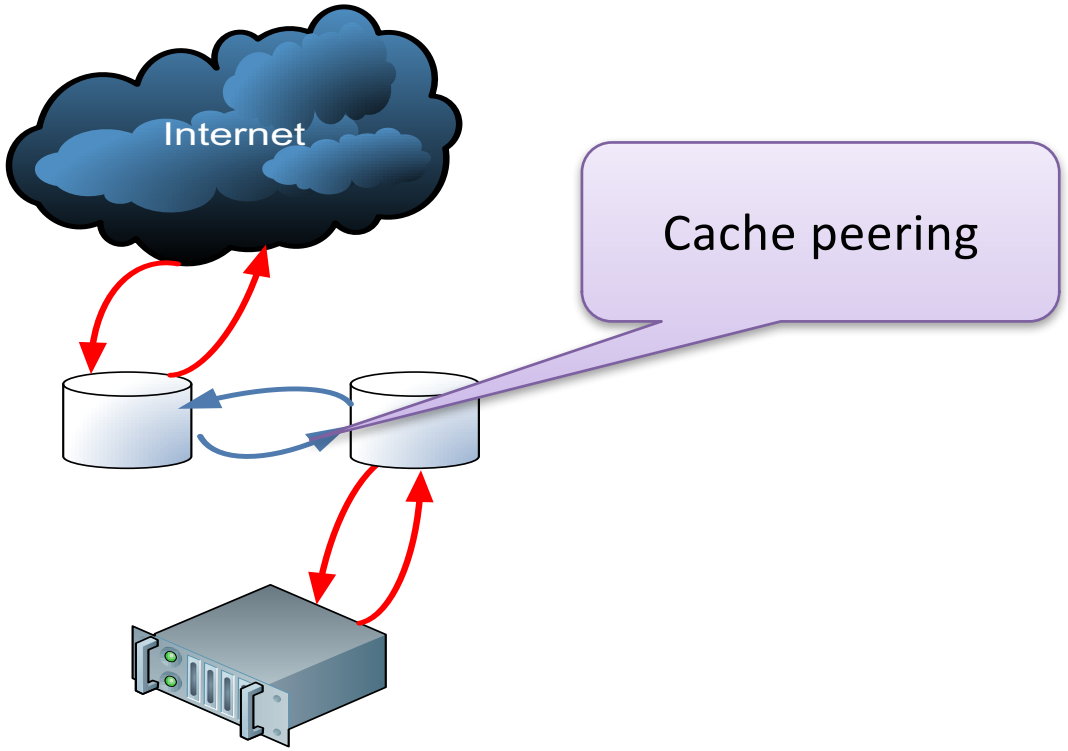
# Validation



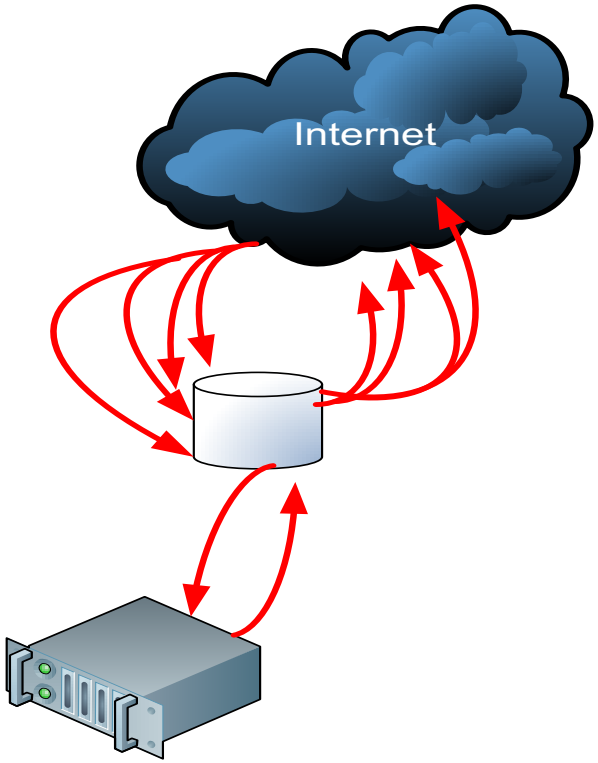
# Scale out!



# Scale the cache out!

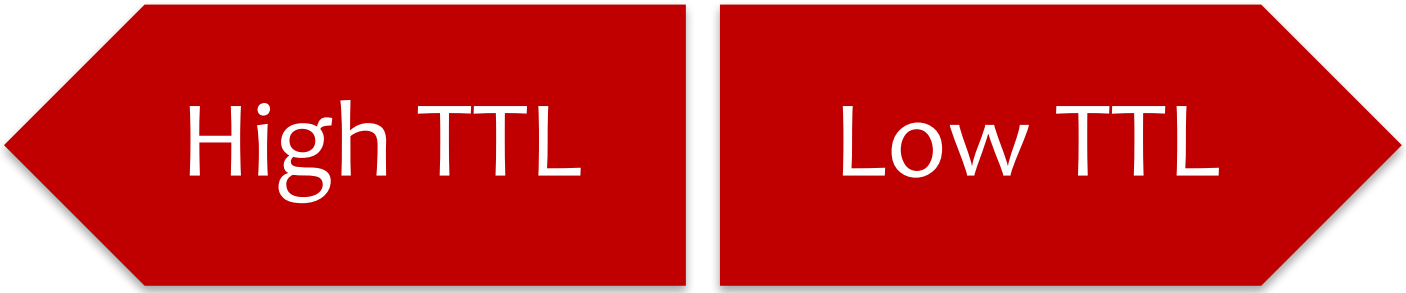


# Collapsed Forwarding



## Caching dilemma

Efficient use of network resources




Publisher controls freshness of data

## Cache channels

Mark Nottingham, Yahoo

- http://www.mnot.net/cache_channels/

Use Atom to extend the freshness of cached responses

Response

```
Cache-Control: max-age=60, channel="http://restbucks.com/products/channel/index",
channel-maxage
```

Response remains fresh as long as:

- Cache polls channel at least as often as "precision" specified by channel
- Channel doesn’t issue stale event

## Being workshy is a good thing!

- Provide guard clauses in requests so that servers can determine easily if there's any work to be done
  - Caches too
- Use headers:
  - `If-Modified-Since`
  - `If-None-Match`
  - And friends
- Web infrastructure uses these to determine if its worth performing the request
  - And often it isn't
  - So an existing representation can be returned

## Conditional `GET` Avoids Work!

- Bandwidth-saving pattern
- Requires client and server to work together
- Server sends `Last-Modified` and/or `ETag` headers with representations
- Client sends back those values when it interacts with resource in `If-Modified-Since` and/or `If-None-Match` headers
- Server responds with a 200 an empty body if there have been no updates to that resource state
- Or gives a new resource representation (with new `Last-Modified` and/or `ETag` headers)

Opaque identifier for resource state

# Retrieving a Resource Representation

Request

```
GET /orders/1234 HTTP 1.1
Host: restbucks.com
Accept: application/vnd.restbucks+xml
If-Modified-Since: 2009-01-08T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-4bca-ba3ef3f50432
```

Response

```
200 OK
Content-Type: application/vnd.restbucks+xml
Content-Length: ...
Last-Modified: 2009-01-08T15:10:32Z
Etag: abbb4828-93ba-567b-6a33-33d374bcad39

<order … />
```

## Not Retrieving a Resource Representation

Request

```
GET /orders/1234 HTTP 1.1
Host: restbucks.com
Accept: application/vnd.restbucks+xml
If-Modified-Since: 2009-01-08T15:00:34Z
If-None-Match: aabd653b-65d0-74da-bc63-
  4bca-ba3ef3f50432
```

Response

```
HTTP/1.1 304 Not Modified
```

Client's representation of the resource is up-to-date

## Works with other verbs too

```
PUT /orders/1234 HTTP 1.1
Host: restbucks.com
Accept: application/vnd.restbucks+xml
If-Modified-Since: 2007-07-08T15:00:34Z
If-None-Match: aabd653b

<order …/>
```

## PUT Results in no Work Done

```
200 OK
Content-Type: application/vnd.restbucks+xml
Content-Length: ...
Last-Modified: 2007-07-08T15:00:34Z
Etag: aabd653b
```


Being workshy is awesome!

## Finally… An Urban Legend


- I cannot say this anecdote is categorically true
  - But I want it to be true so badly…

## Summary

- Statelessness throughout
- Caching everywhere
- Caches are clever
- Control your caching to suit your business need
- Do no work if you can