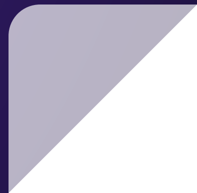
Safe Client Behaviour

Ariel Goh <gwy@google.com>
Sebastian Kirsch <skirsch@google.com>
SRECon18 Asia, Singapore, June 6-8 2018



client
server

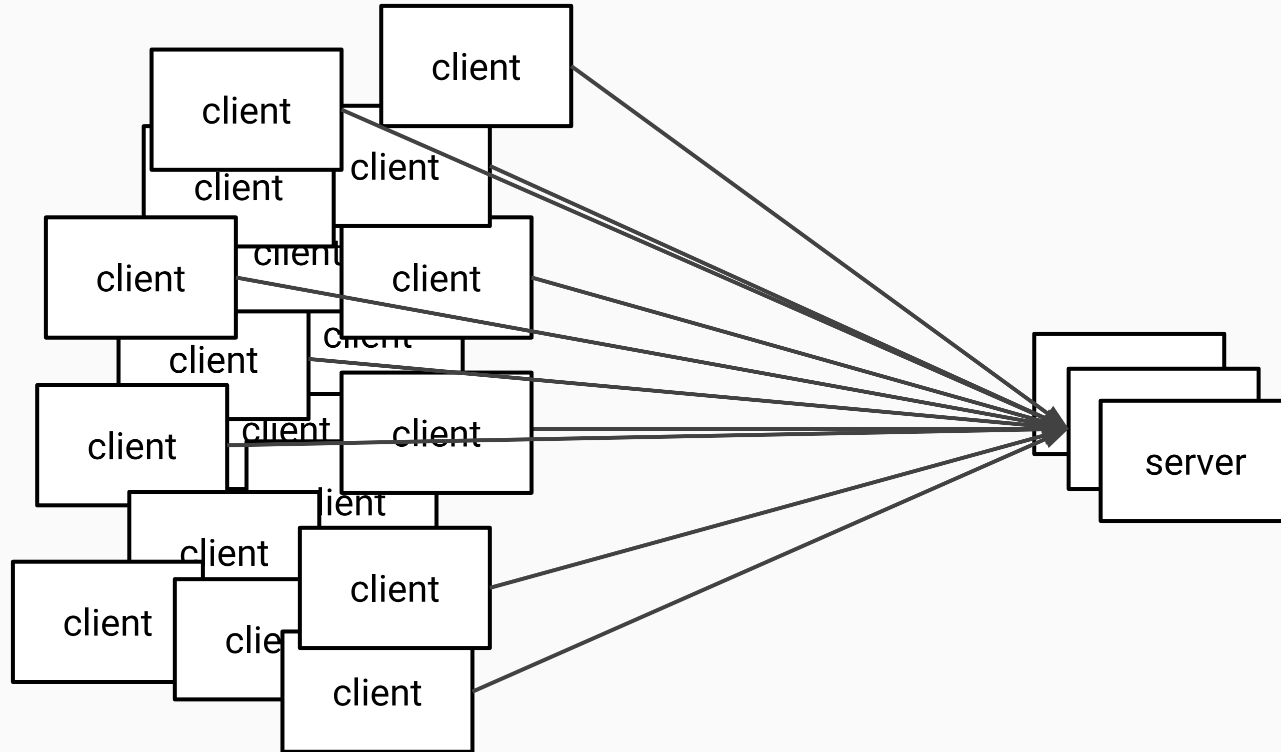
2 Billion

monthly active Android devices

Source: Google announcement, May 2017

2 … 20 … 200?

servers in your cloud service?


client
client
client
client
client
client
client
client
client
client
client
client
client
server

clients ⪢ servers

Safe client behaviour

=

“do not DDoS”

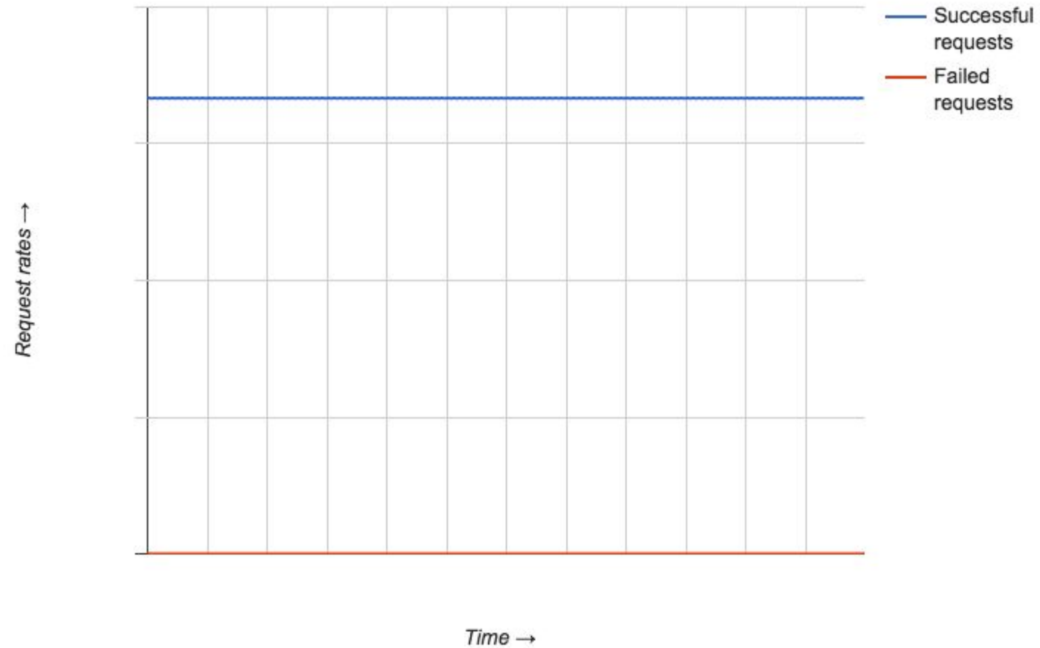
Which requests are potentially unsafe?

What’s the worst that can happen?

Source: https://commons.wikimedia.org/wiki/File:Building_7_fire.jpg Licensed under CC BY-SA 2.5

What’s the *best* that can happen?

Ideal server-side request pattern



Rules for safe clients

Rule 1:
Jitter Everything!

jitter / ˈdʒɪtə/ *n.*

the deviation from true periodicity of a presumably periodic signal.

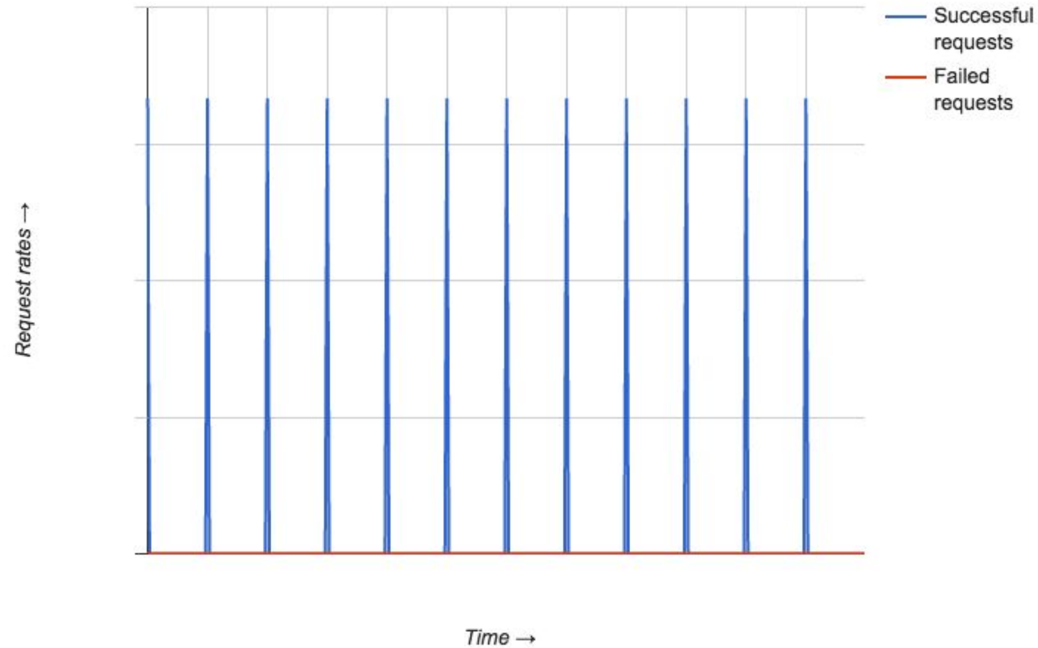
jitter / ˈdʒɪtə/ *v.*

add randomness to the periodicity of a periodic signal.

Worst case: Synchronized startup, no jitter.

```
period = 300 // Once every 5 minutes
while true:
  send_rpc()
  wait(period)
```

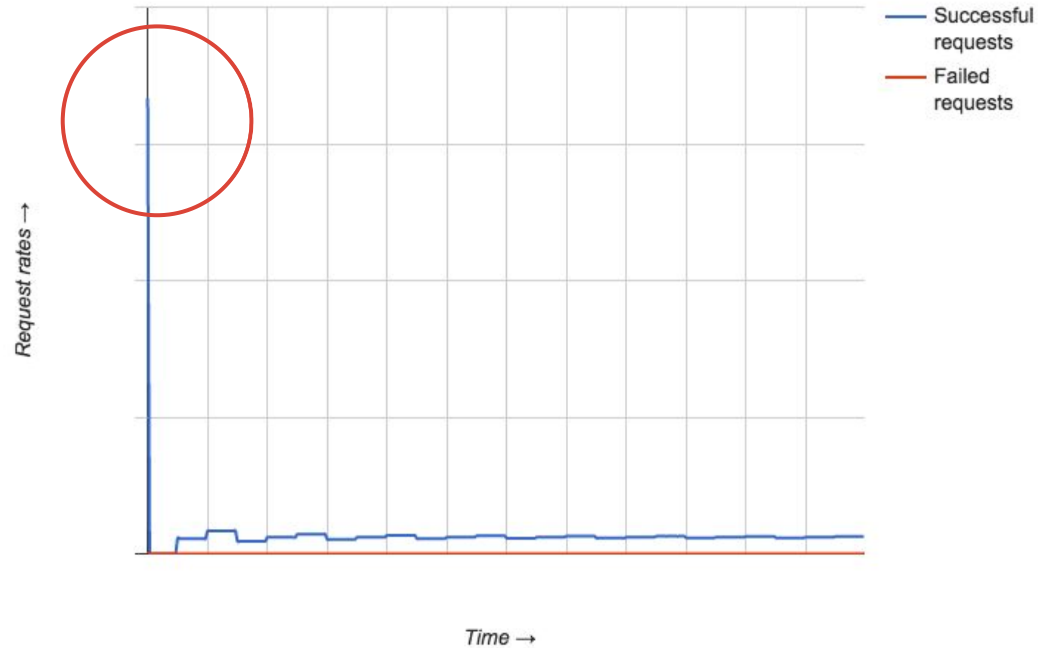
Worst case: Synchronized startup, no jitter.



Injecting jitter: Randomly change the periodicity.

```
period = 300 // Once every 5 minutes
while true:
  send_rpc()
  wait(period * random(.5, 1.5))
```

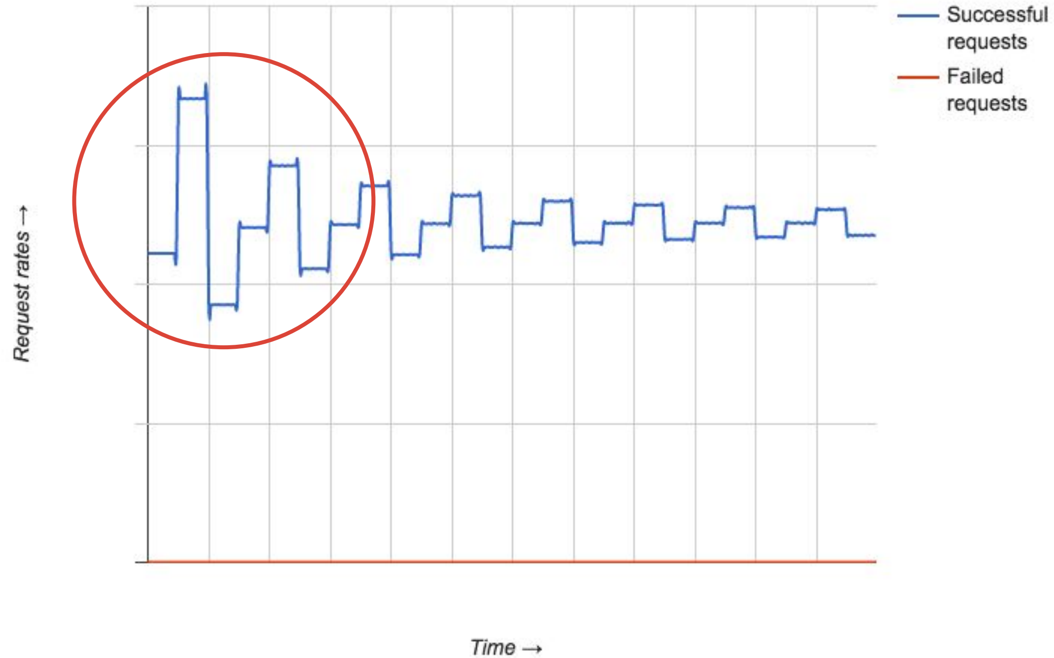
Synchronized startup, jitter on future attempts



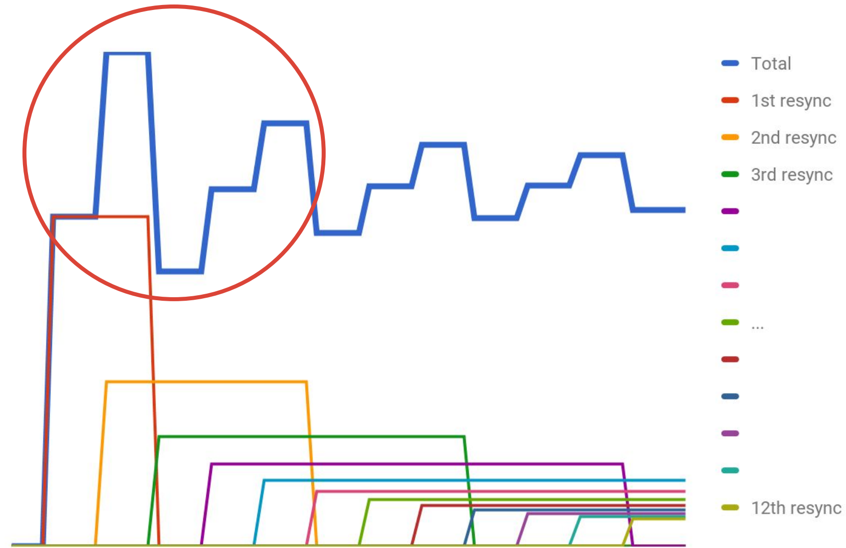
Ideal case: Startup jittered, future attempts jittered.

```
period = 300 // Once every 5 minutes
wait(period * random(.5, 1.5))
while true:
  send_rpc()
  wait(period * random(.5, 1.5))
```

Ideal case: Startup jittered, future attempts jittered.



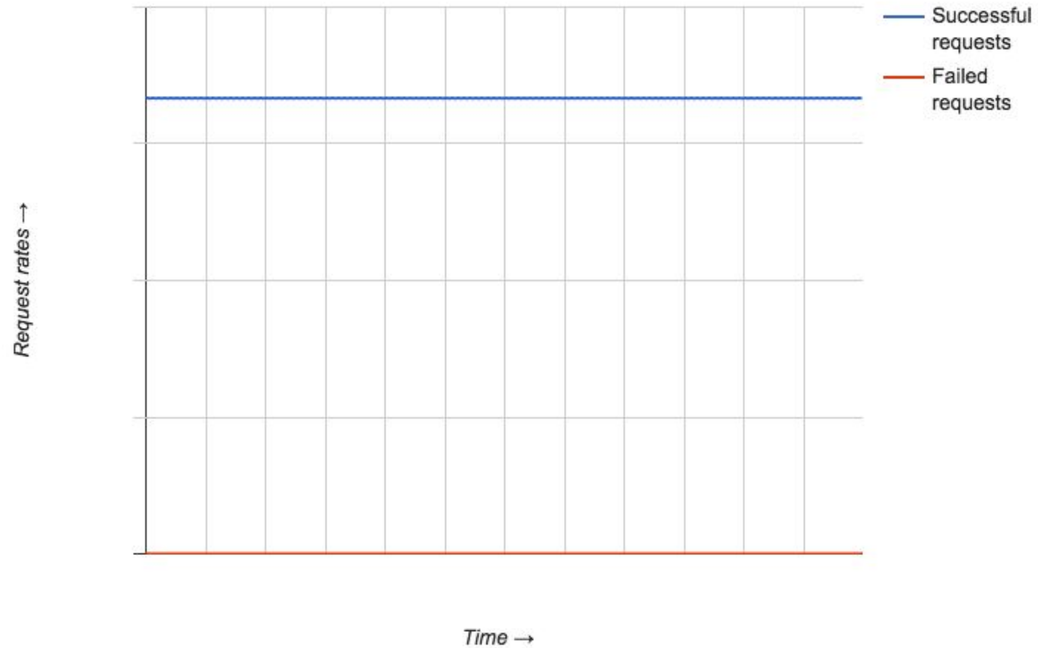
Startup oscillations from jitter.



Jittering without startup oscillations: Jitter execution time, not period.

```
while true:
  period = 300 // Once every 5 minutes
  next_execution = now()
  next_execution = truncate(next_execution, period)
  next_execution += random(1.0, 2.0) * period
  wait_until(next_execution)
  send_rpc()
```

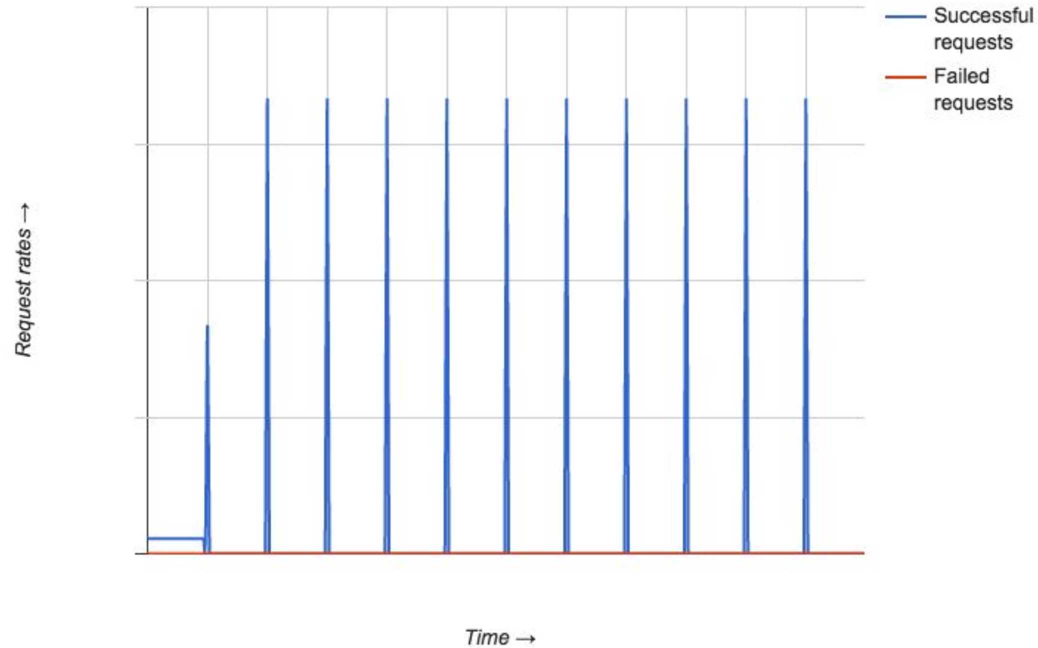
Jittering without startup oscillations: Jitter execution time, not period.



Jittering without startup oscillations: Introducing synchronization by mistake.

```
while true:
  period = 300 // Once every 5 minutes
  next_execution = now()
  next_execution += random(1.0, 2.0) * period
  next_execution = truncate(next_execution, period)
  wait_until(next_execution)
  send_rpc()
```

Jittering without startup oscillations: Introducing synchronization by mistake.



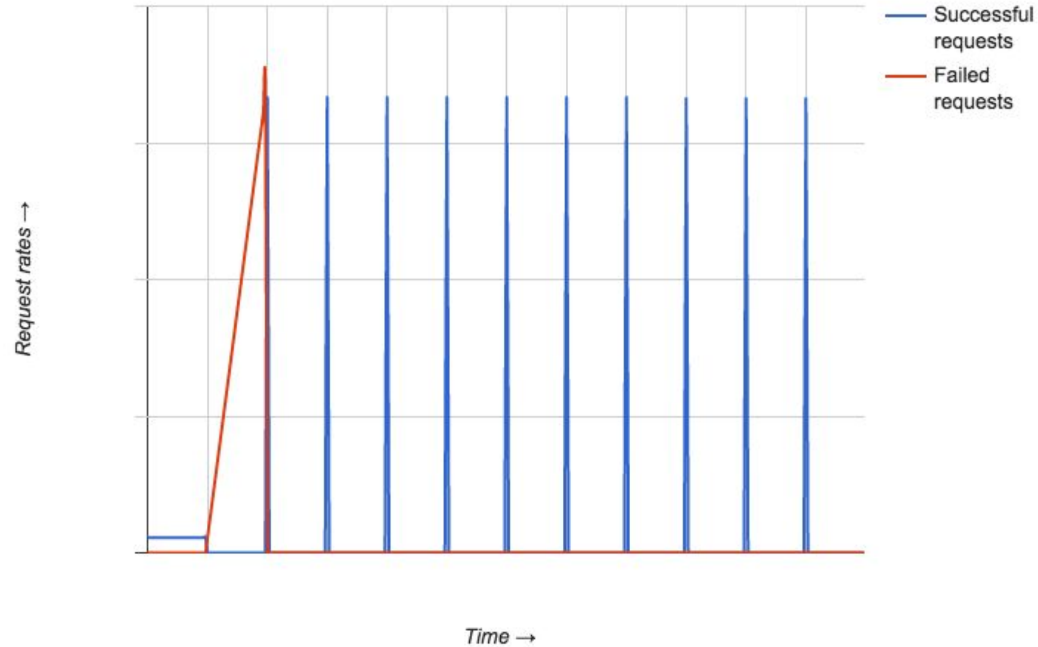
Rule 2:
Don’t Retry!*

* Terms and conditions apply.

Fixed retry period, no jitter.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay)
    success = send_rpc()
  wait(period)
```

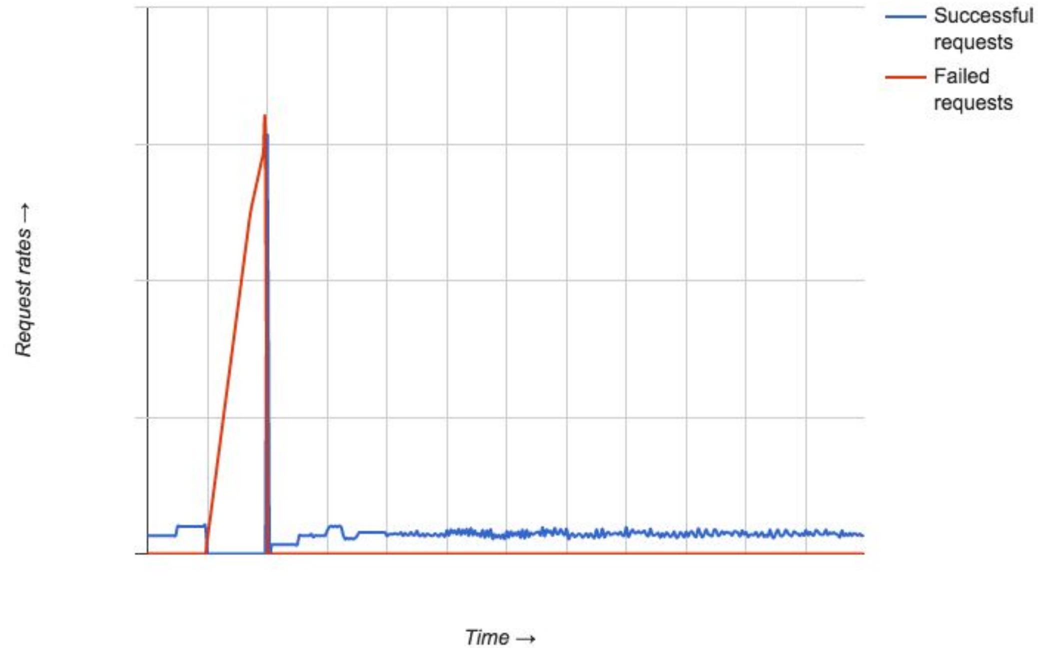
Fixed Retry Period, no jitter: Request spike and inadvertent synchronization



Fixed retry period with jitter.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay)
    success = send_rpc()
  wait(period * random(.5, 1.5))
```

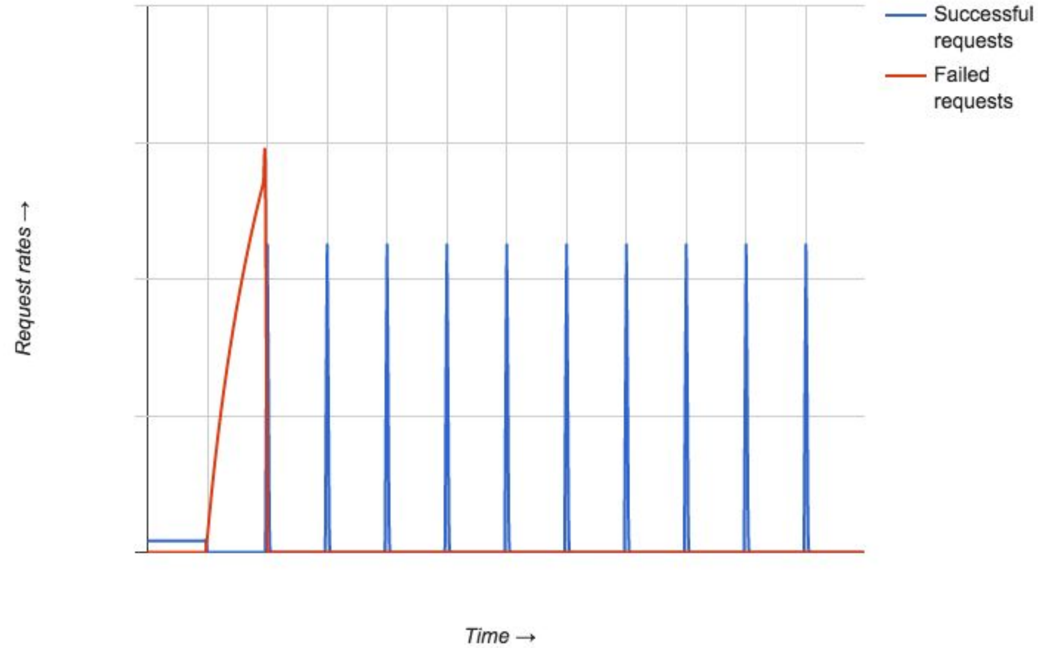
Fixed retry period with jitter: No inadvertent synchronization



Jittered retry period, periodicity not jittered.

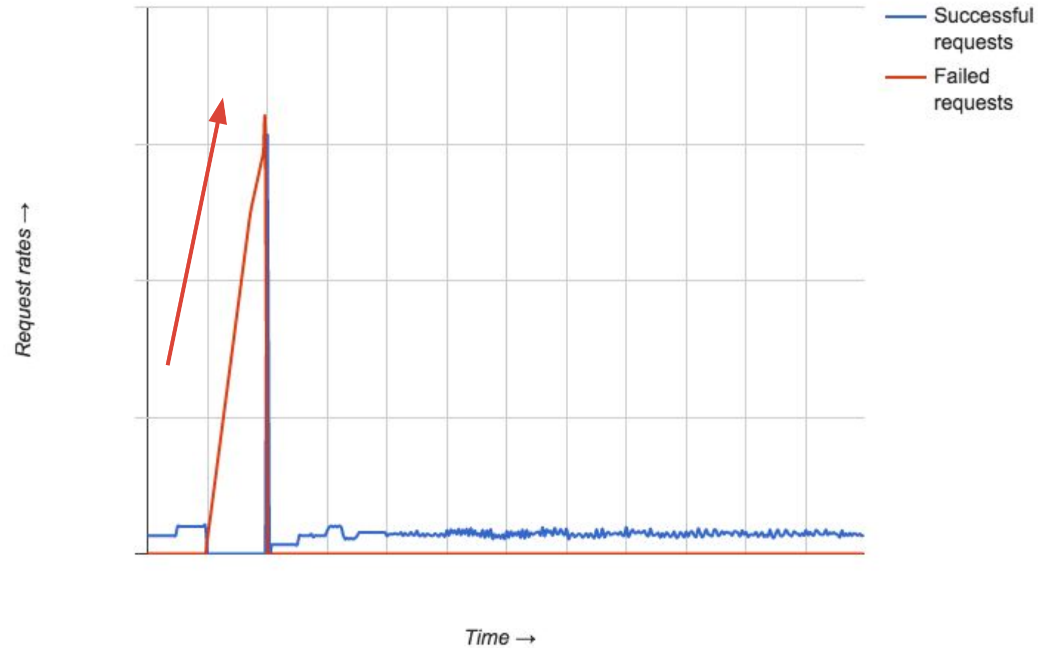
```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay * random(.5, 1.5))
    success = send_rpc()
  wait(period * random(.5, 1.5))
```

Jittered retry period, periodicity not jittered: Request spikes remain.



Rule 3:
If you retry, back off!

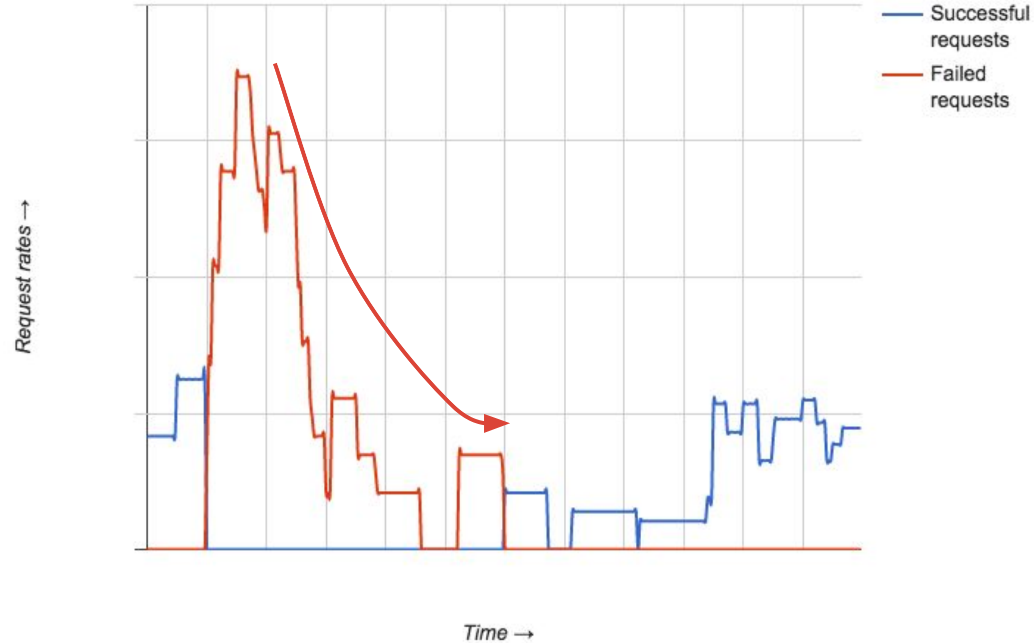
Fixed retry period revisited: Sharp rise in request rate!



Retry with exponential back-off.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay)
    success = send_rpc()
    delay = delay * 2
  wait(period * random(.5, 1.5))
```

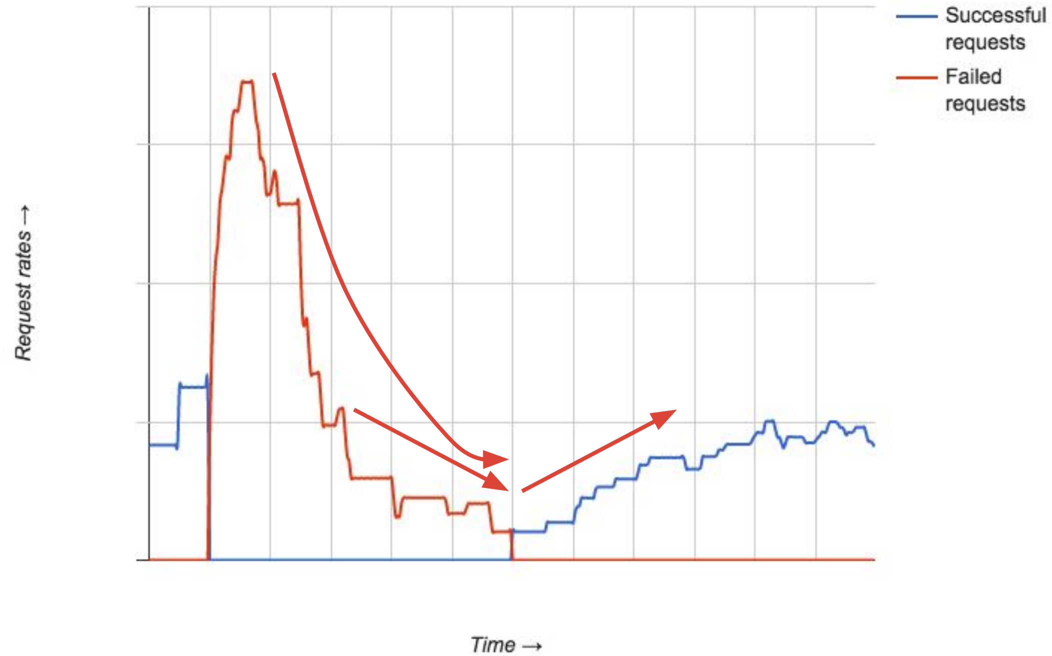
Exponential back-off to limit request spikes.



Retry with exponential back-off and jitter.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay * random(.5, 1.5))
    success = send_rpc()
    delay = delay * 2
  wait(period * random(.5, 1.5)
```

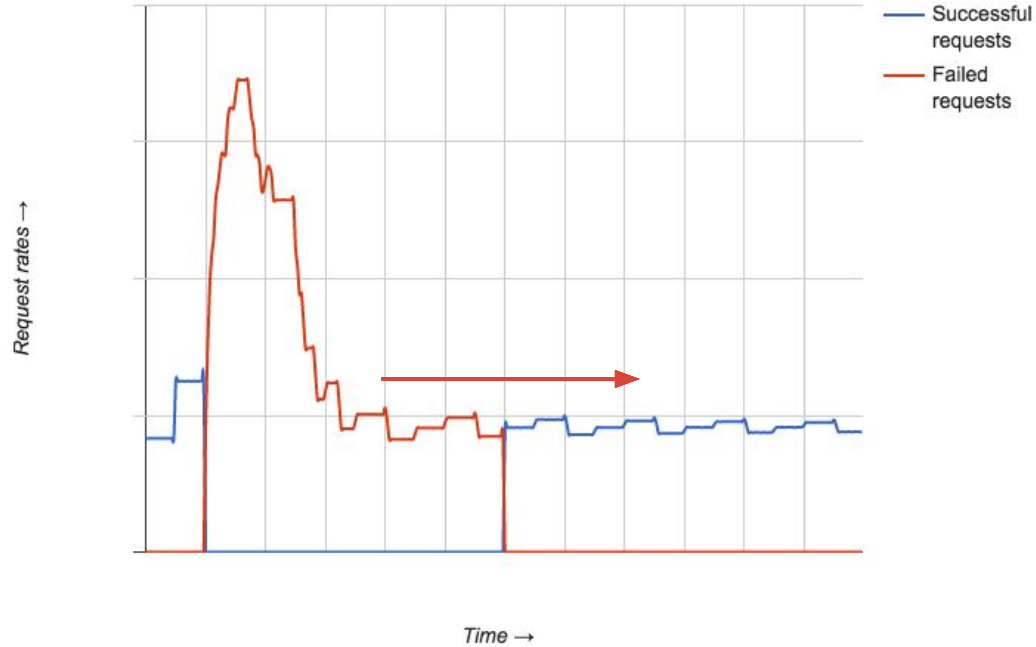
Exponential back-off with jitter: Smoother back-off



Retry with exponential back-off and jitter; retries capped.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success && delay <= period:
    wait(delay * random(.5, 1.5))
    success = send_rpc()
    delay = delay * 2
  wait(period * random(.5, 1.5)
```

Retry with exponential back-off and jitter; retries capped.



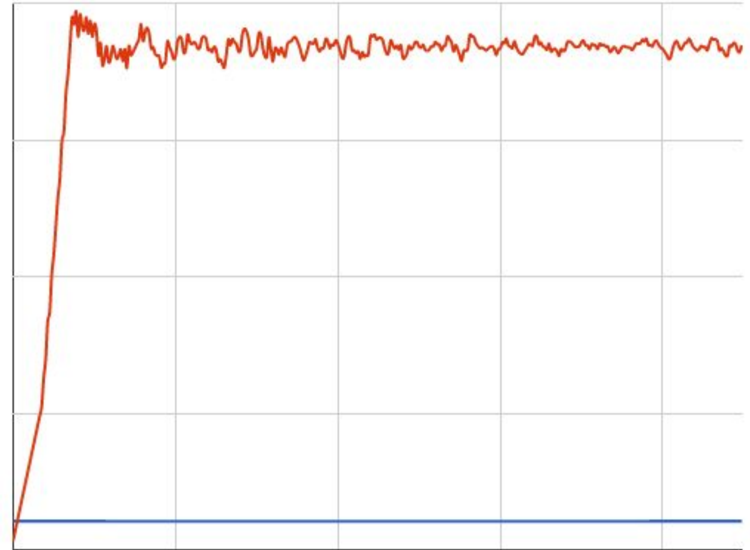
Retries: Terms and Conditions

- Don’t retry by default.
- Back off exponentially on retries.
- Jitter retries.
- Retry on specific error conditions:
  - Don’t retry on client errors (HTTP 400 errors.)
  - Do retry on server errors (HTTP 500 errors.)
  - Do retry on network errors.
  - Do retry (carefully) on timeouts.
  - Don’t retry on out of quota!

Effects of partial failures

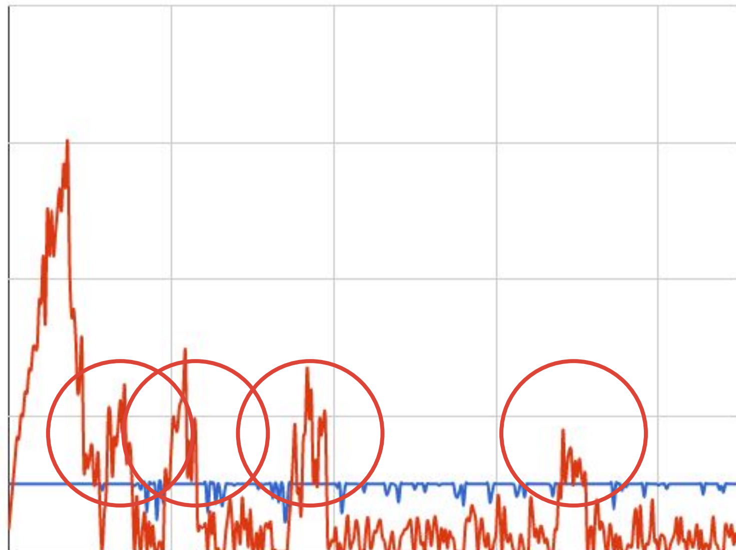
25% failure rate, fixed retry period: High rate of retries

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay)
    success = send_rpc()
  wait(period)
```



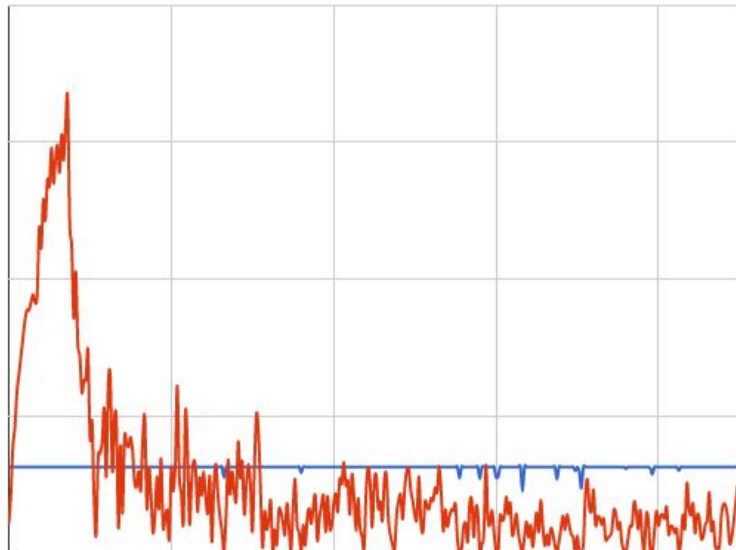
25% failures with exponential back-off: Error rate subsides, occasional error spikes.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay)
    success = send_rpc()
    delay = delay * 2
  wait(period * random(.5, 1.5))
```



25% failure, jittered back-off: Fewer synchronized errors spikes.

```
while true:
  period = 300; delay = 10
  success = send_rpc()
  while not success:
    wait(delay * random(.5, 1.5))
    success = send_rpc()
    delay = delay * 2
  wait(period * random(.5, 1.5)
```



Safer clients: Move control to the server!

Implement `Retry-After` header in client and server.

Make sure to jitter retry periods on the server side.

Remote configure your clients!

Configure periodicity remotely.

Maintain a remotely configured client feature blacklist.

Safer clients: Expose information to server

- More information at server = more granular responses
- Tag requests with
  - Client name and version
  - Feature that triggered the request
  - Severity of failing the request
  - Retry or initial request?
- Possible server responses:
  - Prioritize interactive requests
  - Drop background requests
  - Avoid dropping requests known to trigger a retry storm
  - Work around client bugs

Safer Microservices

Safer microservices: Retry budgets

Limit retries to a percentage of total outgoing requests

Example: Only 10% of outgoing requests can be retries.

Prevents retries from completely crowding out legitimate requests.

Safer microservices: Adaptive Throttling

Reject new requests locally (at the client) based on rejection probability.

Reduces load on the server for rejected requests.

See https://landing.google.com/sre/book/chapters/handling-overload.html

Summary

1. Jitter everything!
2. Don’t retry!
3. If you retry, back off!
4. Move control to the server.
5. Expose info to the server.
6. Use retry budgets.
7. Use adaptive throttling.

Q&A

See also https://landing.google.com/sre/book/chapters/handling-overload.html