O'REILLY®

Software Architecture

ENGINEERING THE FUTURE OF SOFTWARE

softwarearchitecturecon.com

#oreillysacon

Optimizing Uptime in SOA

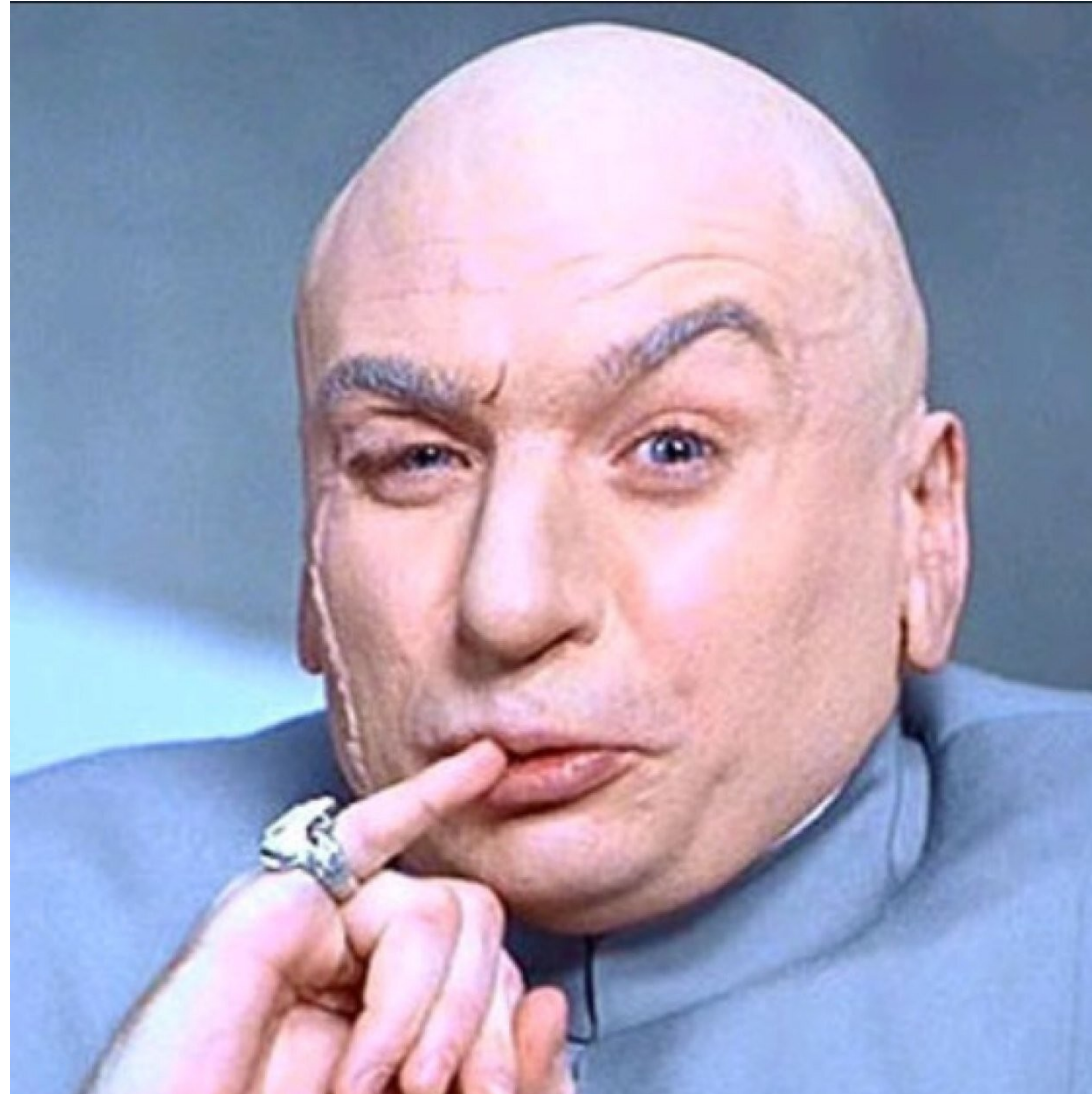
by Matthew Barlocker
Chief Architect @ Lucid Software Inc

“A distributed system is, at best, a necessary **evil**; evil because of the extra complexity...

An application is rarely, if ever, intrinsically distributed. Distribution is just the lesser of the many evils, or perhaps better put, a sensible engineering decision given the trade-offs involved.”

-David Cheriton

Distributed Systems



Local Functions are Deterministic

- Either the function returns or the server shut down
- No network hops
- Availability of libraries is guaranteed if the server is on

"My frontside bus fails for only 1 second every 27.8 hours"

- no one ever

Distributed Systems are Evil

- Network Failure
  - Dropped Packets
  - Corrupted Packets
  - Failed Router
  - Traffic Spike
  - Idiot with a shovel
- Service Not Running
  - Deployment
  - Troubleshooting
  - Rogue Ops Member
- Sick Server
  - Thread starvation
  - Swapping
  - Open file limit
  - Application Monitoring went AWOL

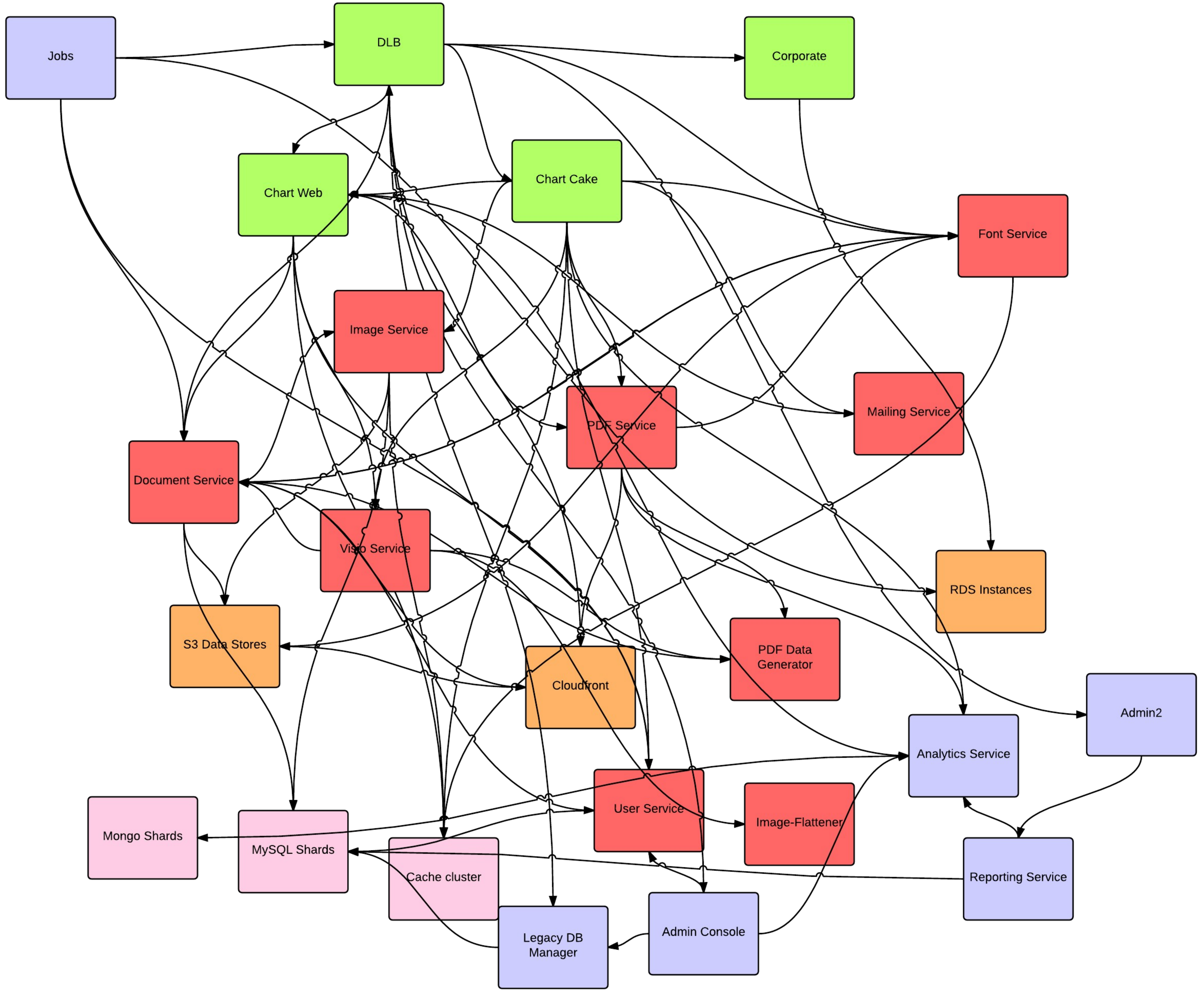
Fallacies of Distributed Systems

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Distributed Systems have Good

- Separation of concern in the organization
- Fail independently
- Resource consumption
- Scaling profiles
- Code modularity
- Loose coupling

Lucid Software has SOA



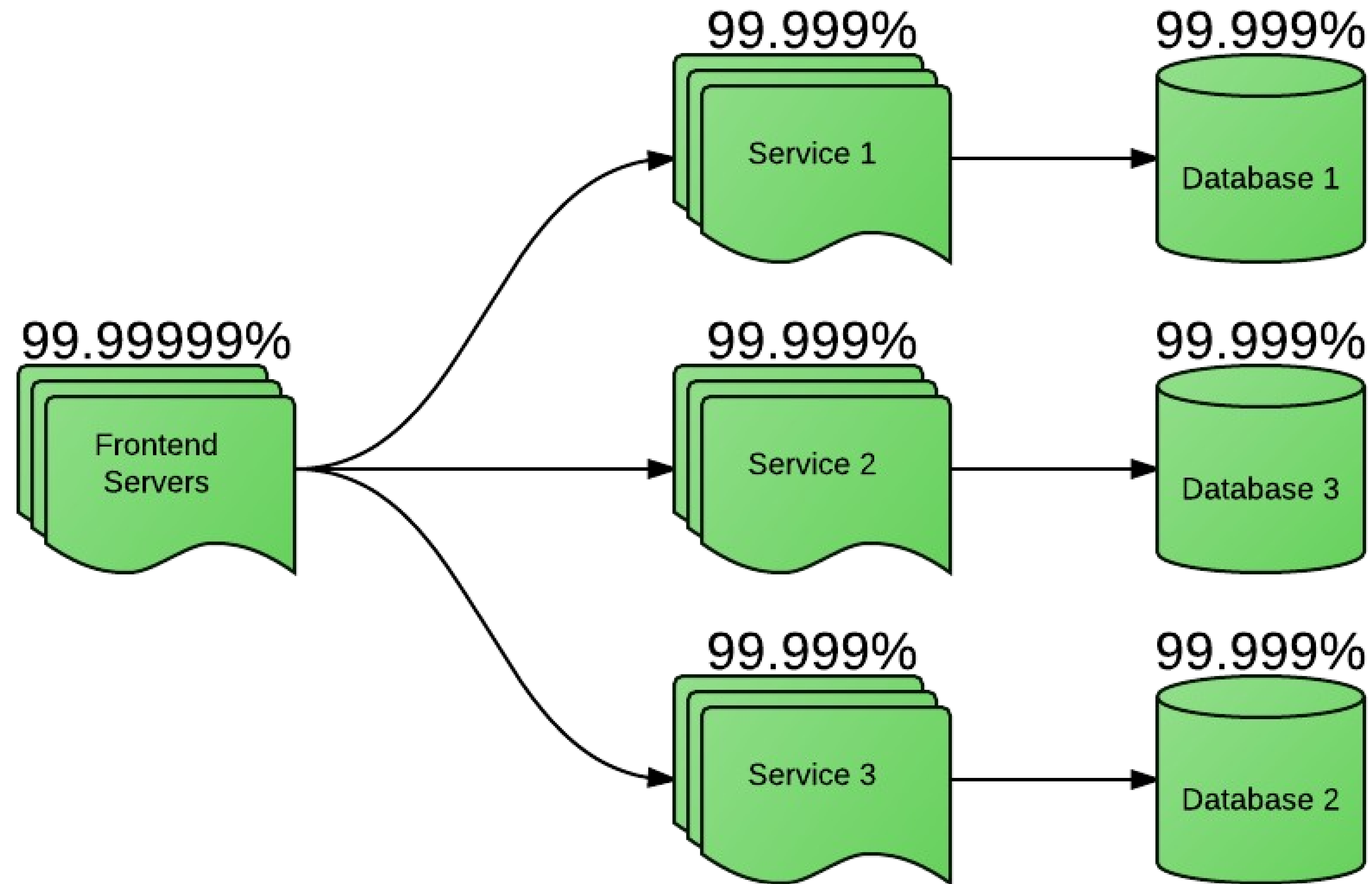
More Parts = More Failures

More Parts = More Failures

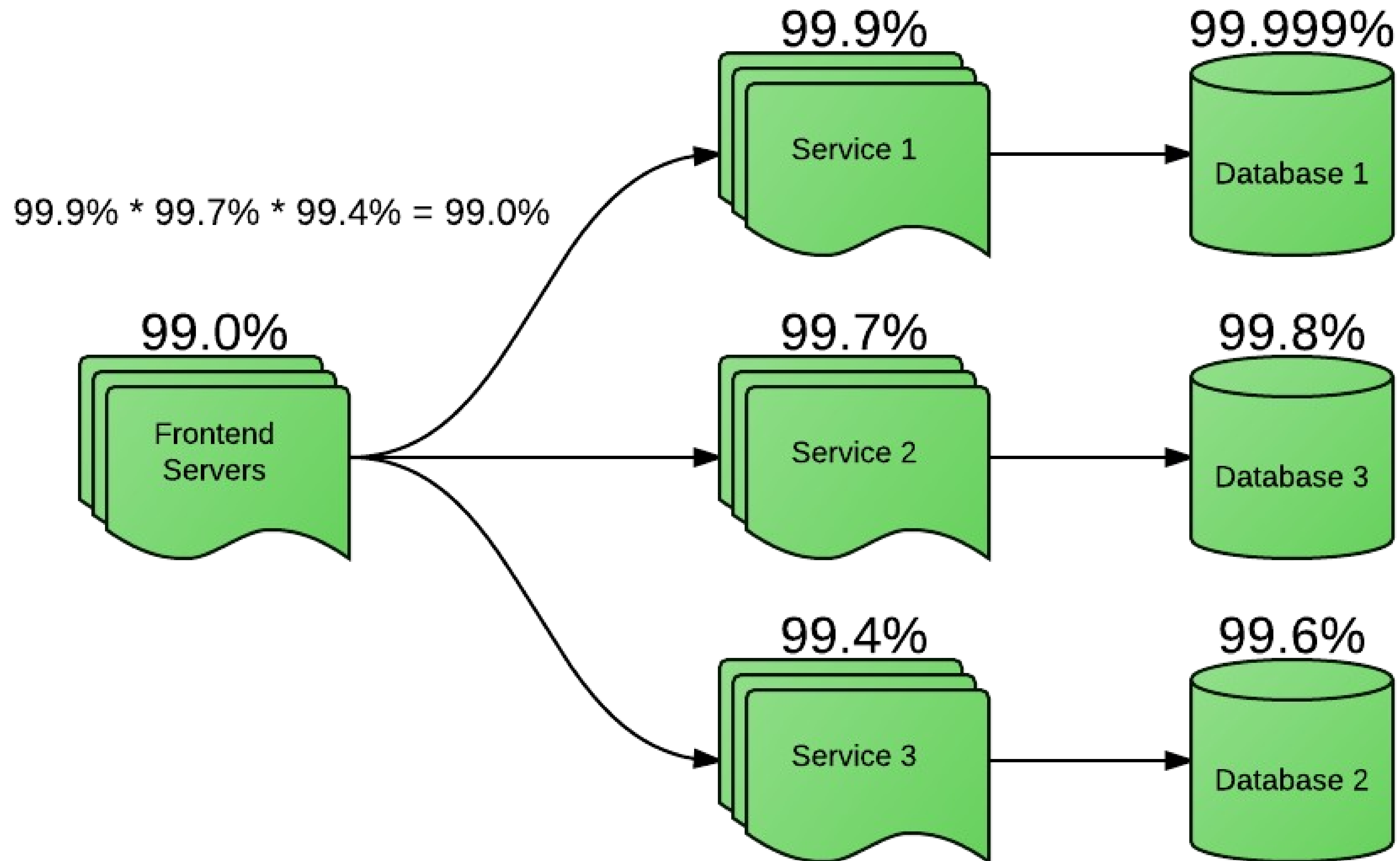
The Domino Effect

- Thread exhaustion
- Memory utilization
- File handle limits
- Database record locks

Uptime – What We Wish For

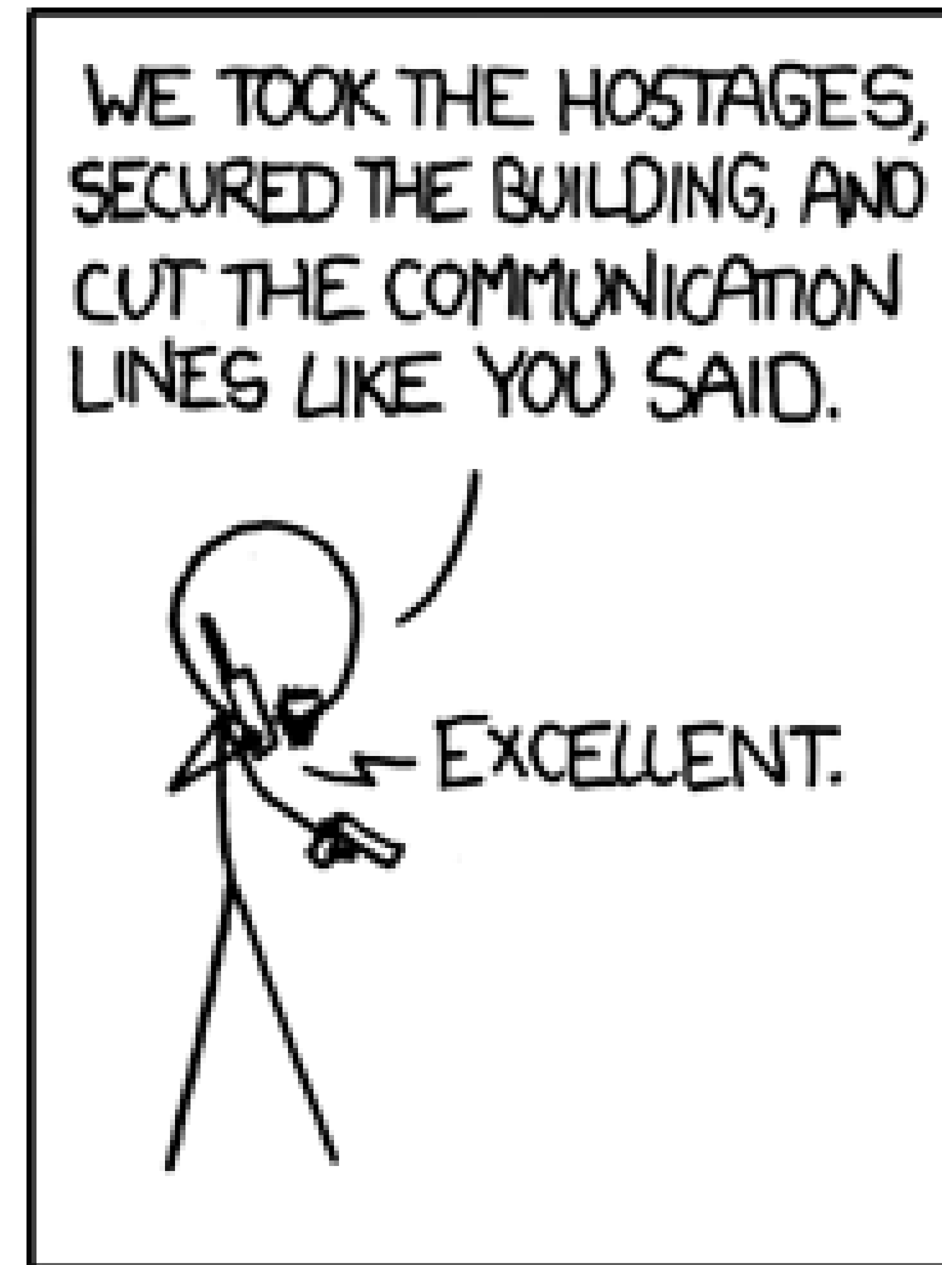


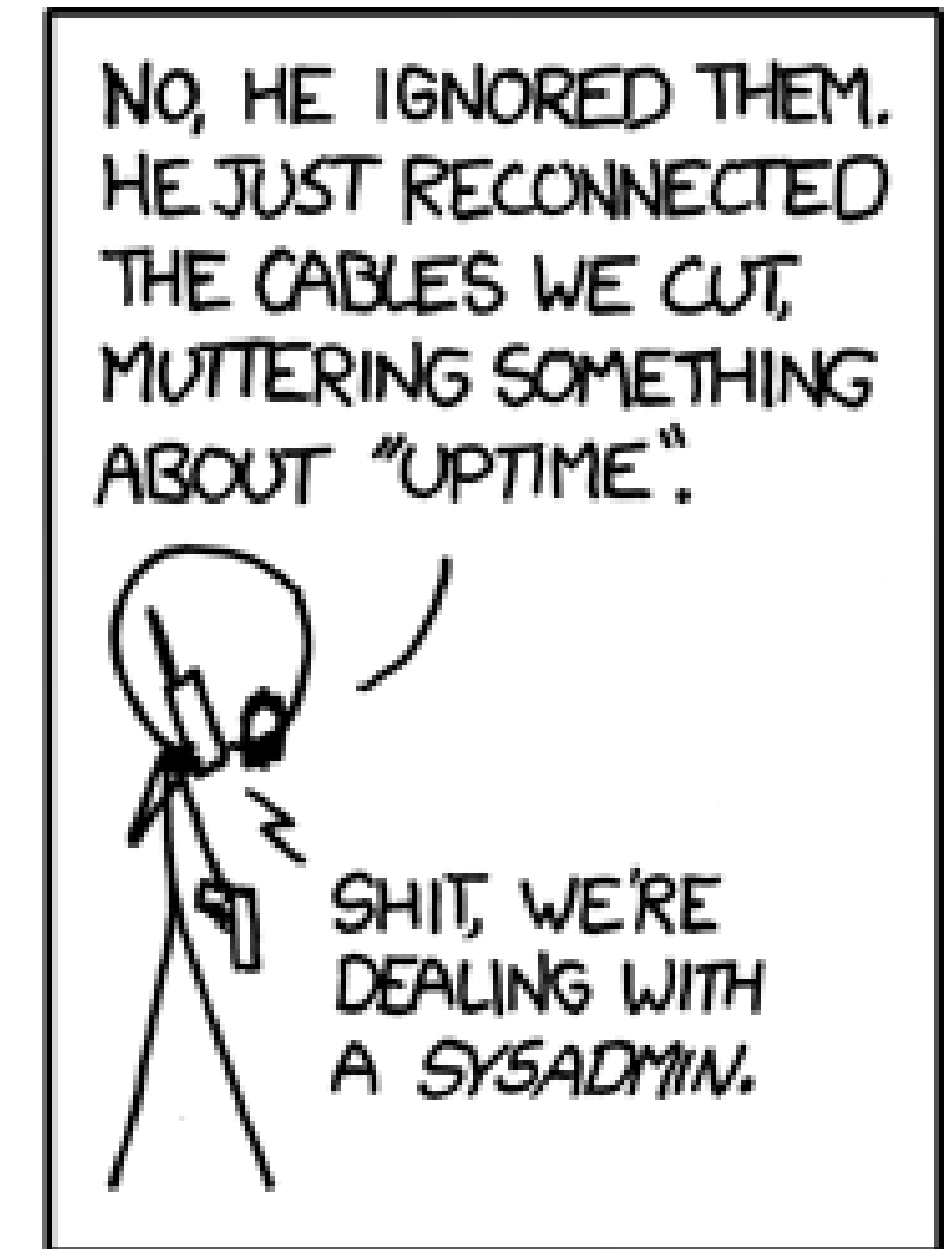
Uptime – What We Get



Strategies for Optimizing Uptime

- Tolerate Failures
  - Timeouts
  - Retries
  - Idempotence
- Sacrifice Consistency
  - Caching
  - Degraded Mode

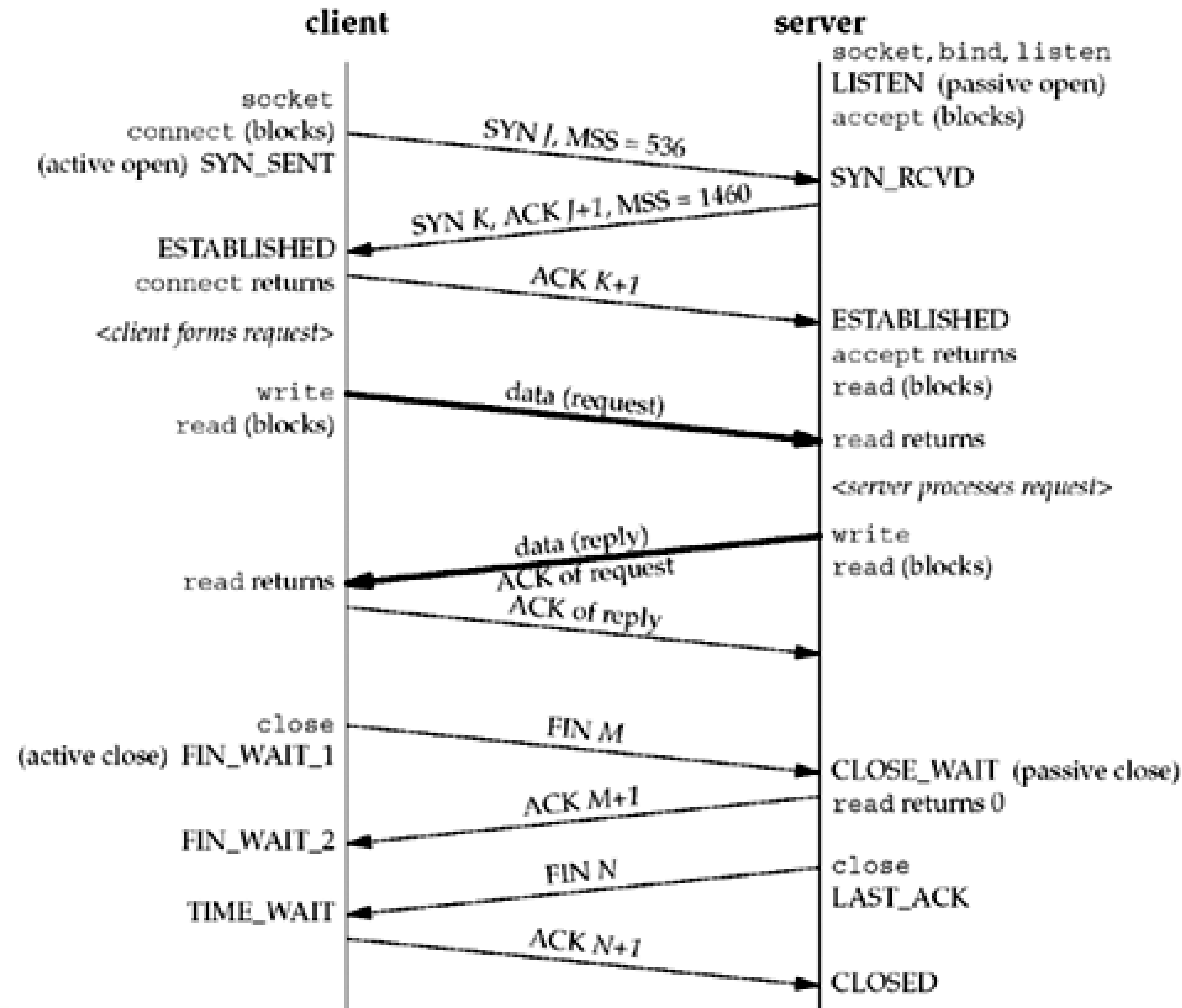




Tolerating Failures

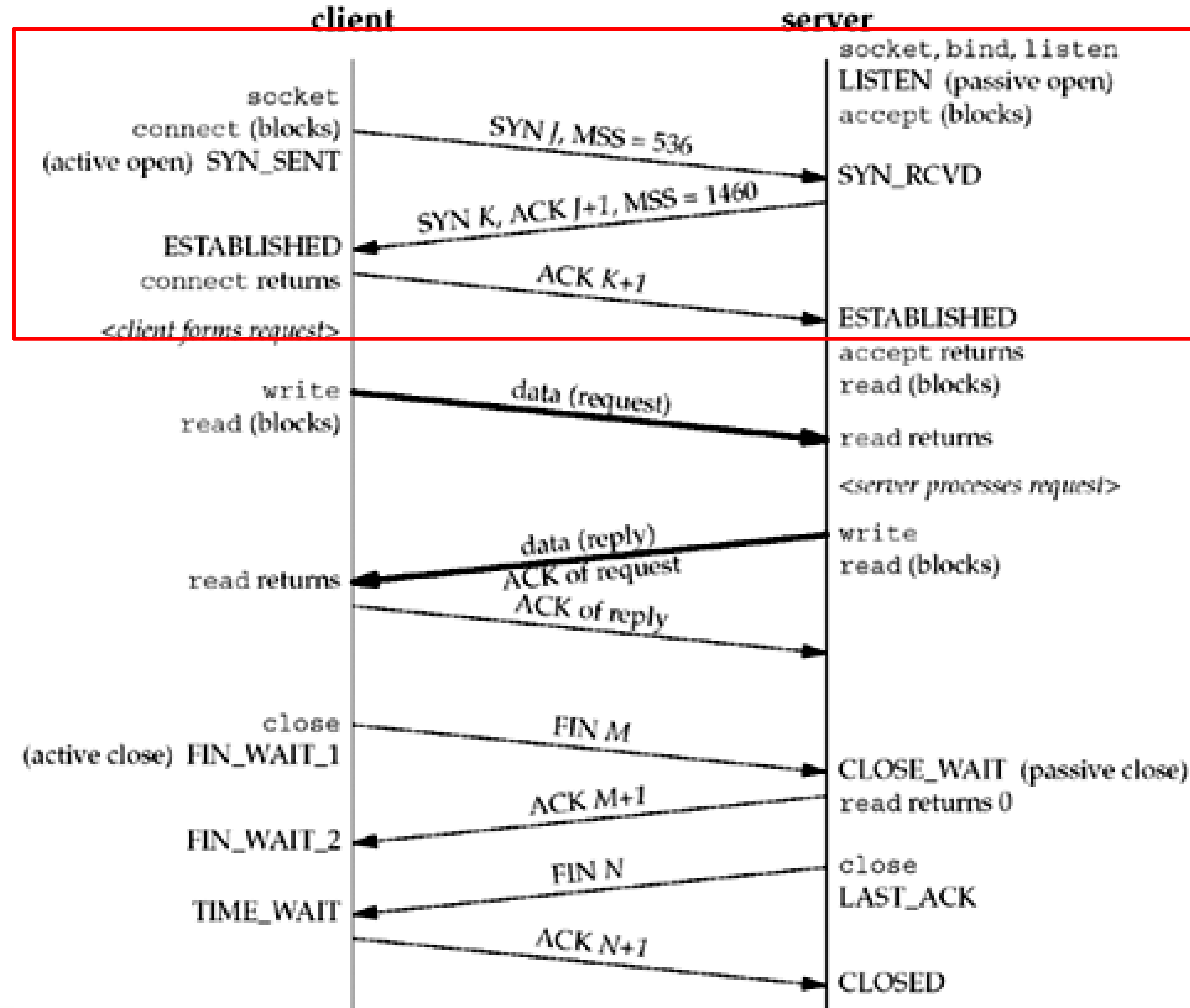
"Just Like Mom"

Timeouts - TCP



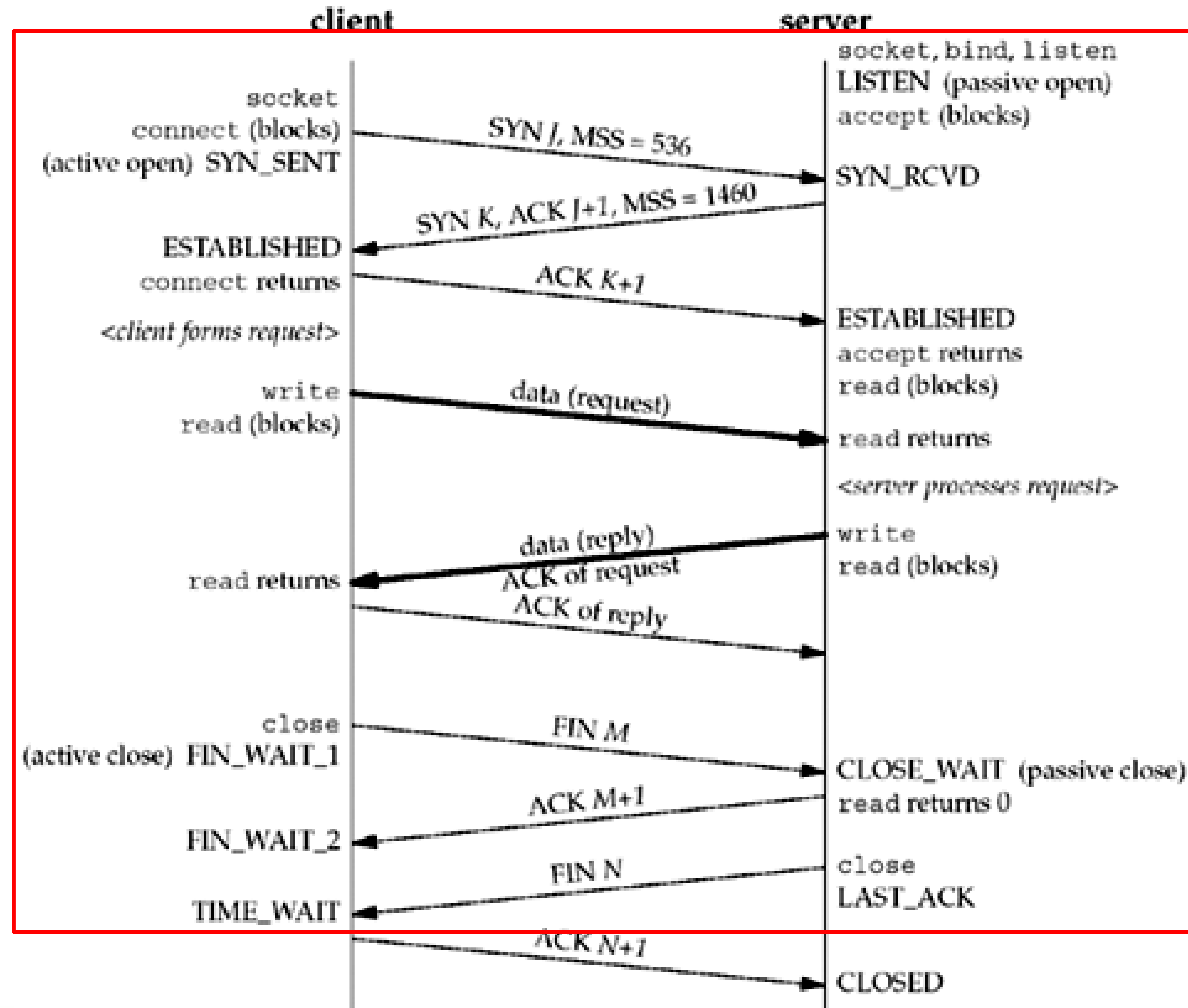
Timeouts - TCP

Connection Timeout



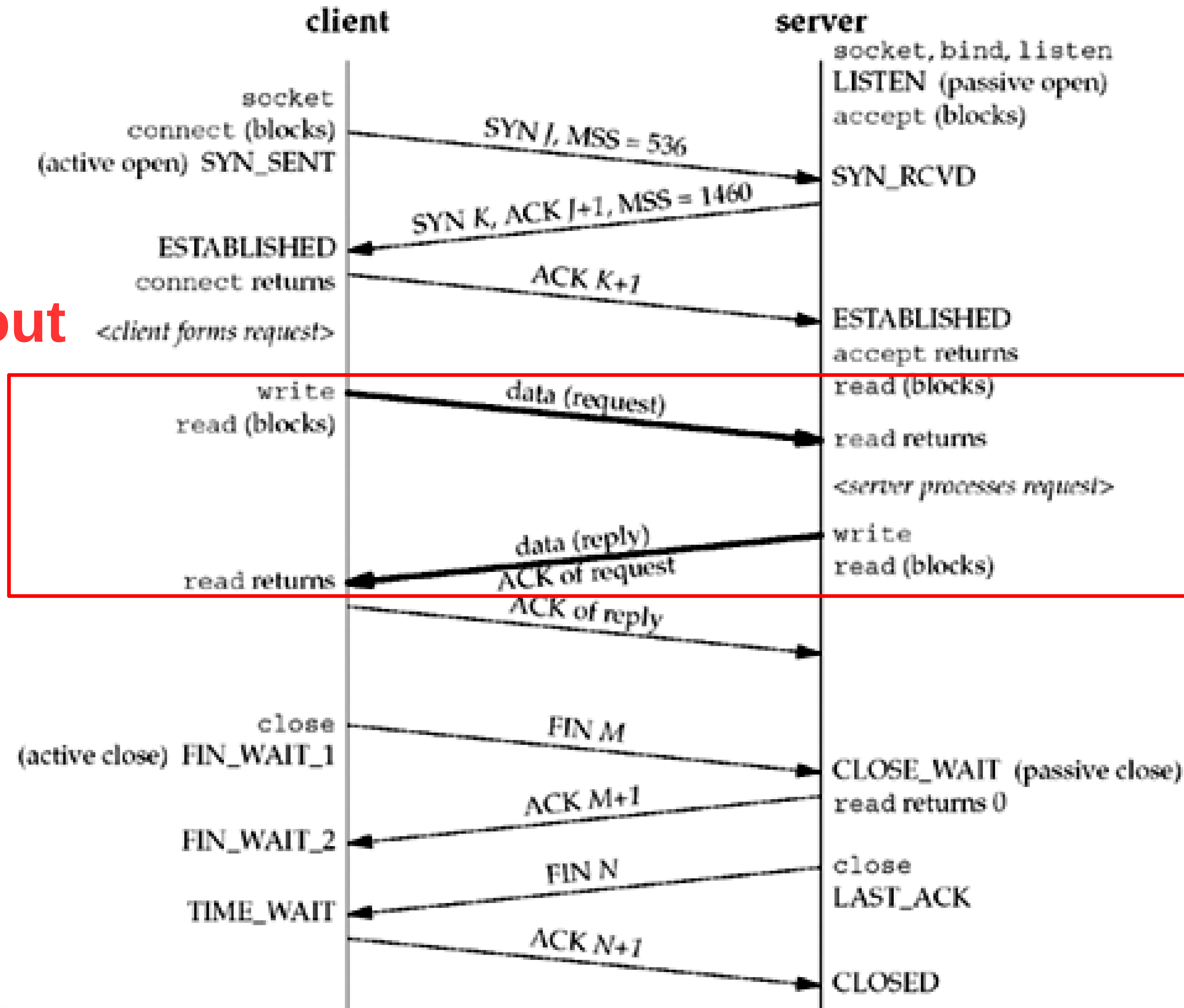
Timeouts - TCP

Lifetime Timeout



Timeouts - TCP

Read Timeout

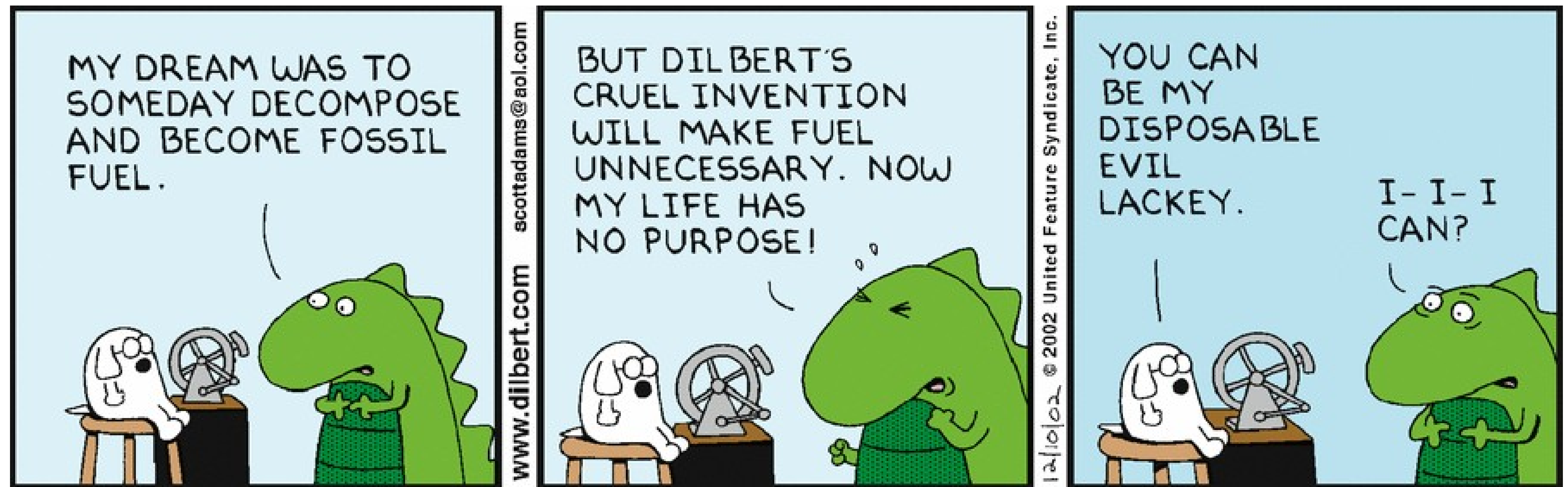


Read timeout resets after each read. Chunked responses rarely get timed out on reads.

O'REILLY® Software Architecture

The Reason to use Timeouts

- Is NOT to limit I/O wait time
- Is NOT to catch theoretical upper bounds
- Is NOT to be a cool unused feature
- Is NOT to be a last resort for user response time



The reason to use timeouts is to increase uptime

by failing fast and allowing retries, caching, and degraded modes.

Timeout Statistics – Around the World (in ms)

From Virginia To:	Min	Max	Avg	Median	Std Dev.	84.00%	97.50%	99.85%
Oregon	59.21	166	68.28	68.96	6.16	74.44	80.6	86.76
California	74.74	185.85	83.47	82.1	9.28	92.75	102.03	111.31
Ireland	76.14	85.81	76.36	76.31	0.22	76.58	76.8	77.02
Brazil	123.53	1089.09	124.04	123.75	8.59	132.63	141.22	149.81
Australia	220.27	339.97	240.99	240.5	9.69	250.68	260.37	270.06

Timeout Statistics – In the Data Center (in ms)

From	To	Min	Max	Average	Median	Std Dev.	84.00%	97.50%	99.85%
us-east-1c	**us-east-1e**	0.4	7.35	0.51	0.49	0.25	0.76	1.01	1.26
us-east-1c	**us-east-1d**	1.39	8.21	1.47	1.45	0.21	1.68	1.89	2.1
us-east-1e	**us-east-1c**	0.41	6.84	0.5	0.5	0.17	0.67	0.84	1.01
us-east-1e	**us-east-1d**	0.71	4.86	0.76	0.75	0.17	0.93	1.1	1.27
us-east-1d	**us-east-1c**	1.4	8.93	1.58	1.52	0.44	2.02	2.46	2.9
us-east-1d	**us-east-1e**	0.71	6.9	0.83	0.81	0.34	1.17	1.51	1.85

Timeout Comparison on Healthy Service

- Low Timeout Version
  - 3ms TCP connect timeout
  - 100ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 1ms – TCP connection established
  - 51ms – Data returned from service

- High Timeout Version
  - 3000ms TCP connect timeout
  - 10000ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 1ms – TCP connection established
  - 51ms – Data returned from service

Both versions are the same when the service is healthy

Timeout Comparison with a Dropped Packet

- Low Timeout Version
  - 3ms TCP connect timeout
  - 100ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 3ms – Make request to service
  - 4ms – TCP connection established
  - 54ms – Data returned from service

- High Timeout Version
  - 3000ms TCP connect timeout
  - 10000ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 1000ms – TCP retransmits packet
  - 1001ms – TCP connection established
  - 1051ms – Data returned from service

With a dropped packet, low timeouts provide 95% speedup

Timeout Comparison with Sick Server

- Low Timeout Version
  - 3ms TCP connect timeout
  - 100ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 1ms – TCP connection established
  - 101ms – Make request 2 to service
  - 102ms – TCP connection established
  - 152ms – Data returned from service

- High Timeout Version
  - 3000ms TCP connect timeout
  - 10000ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 1ms – TCP connection established
  - 1001ms – Make request 2 to service
  - 1002ms – TCP connection established
  - 1052ms – Data returned from service

With a sick server, low timeouts provide 86% speedup

Timeout Comparison with Network Hiccup

- Low Timeout Version
  - 3ms TCP connect timeout
  - 100ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 3ms – Make request 2 to service
  - 4ms – TCP connection established
  - 54ms – Data returned from service

- High Timeout Version
  - 3000ms TCP connect timeout
  - 10000ms TCP read timeout
  - 1 Immediate retry
- Timeline
  - 0ms – Make request to service
  - 4ms – TCP connection established
  - 54ms – Data returned from service

With a network hiccup, there is no difference.

Timeouts are worthless without retries, caching, or some other recovery method.

Retries

Retry Logic can Depend on Scenario

- Retry
  - HTTP 5XX server errors
  - TCP connection timeouts
- Don't Retry
  - HTTP 4XX request errors
- You Decide
  - TCP read timeout
  - HTTP 2XX successes with unrecognizable or unparsable body

Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service					
Stampede effect					
Resource starvation					
User visible slowness					
Once in a blue moon errors					

Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service		X			
Stampede effect					
Resource starvation					
User visible slowness					
Once in a blue moon errors					

Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service		X			
Stampede effect				X	
Resource starvation					
User visible slowness					
Once in a blue moon errors					

Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service		X			
Stampede effect				X	
Resource starvation	X		X		
User visible slowness					
Once in a blue moon errors					

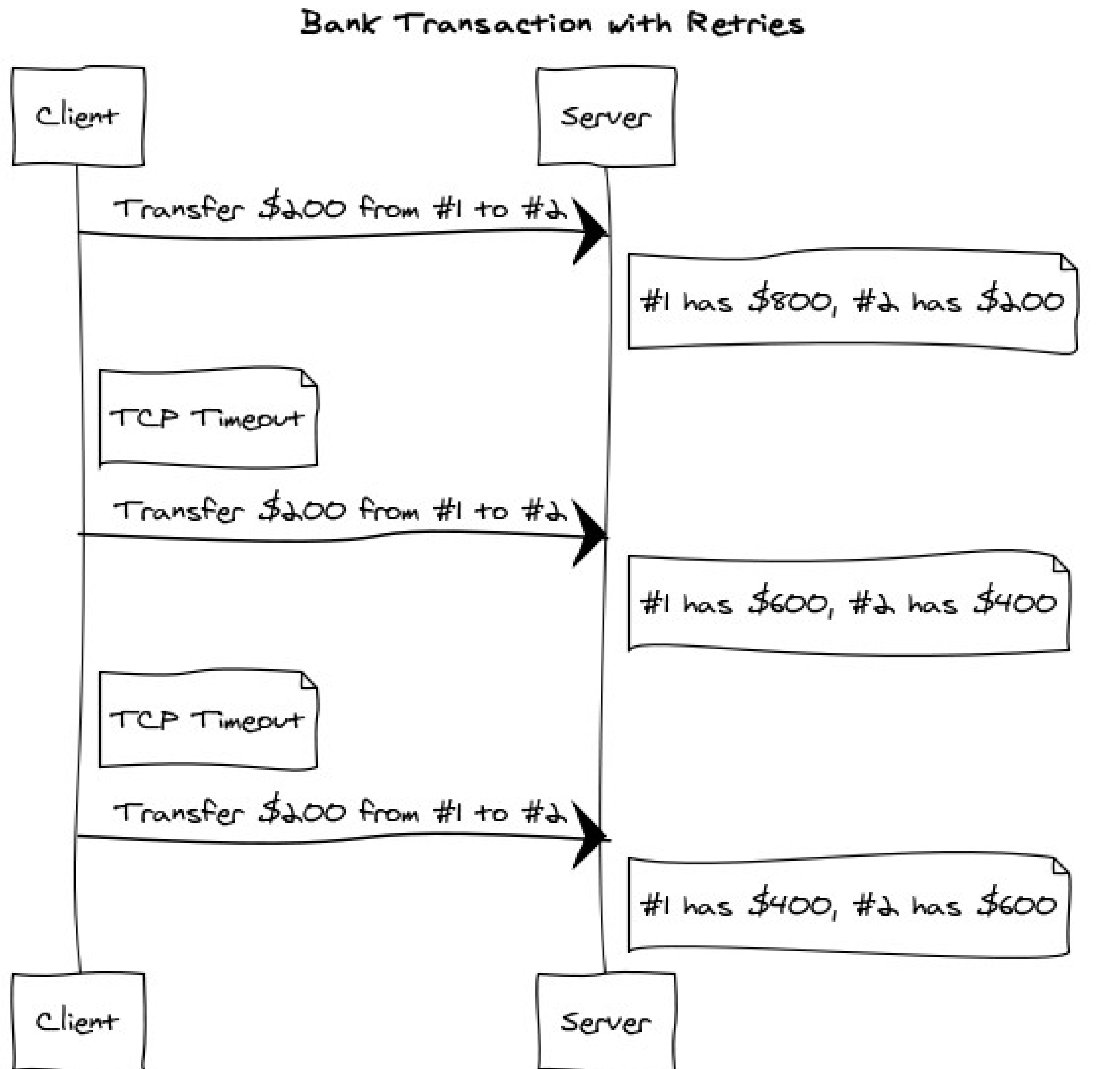
Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service		X			
Stampede effect				X	
Resource starvation	X		X		
User visible slowness	**X**				**X**
Once in a blue moon errors					

Retrying Introduces Solvable Problems

	Low Timeouts	Exponential backoff	Thread pools	Splay	Logging & Stats
Denial of service		X			
Stampede effect				X	
Resource starvation	X		X		
User visible slowness	X				X
Once in a blue moon errors					X

Retrying can Corrupt Data



- Transferring money between bank accounts
- Registering a user
- Add a block to a document
- Delete 10 oldest users on account
- Replace current API credentials
- ...

Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.

O'REILLY®
Software Architecture
ENGINEERING THE FUTURE OF SOFTWARE

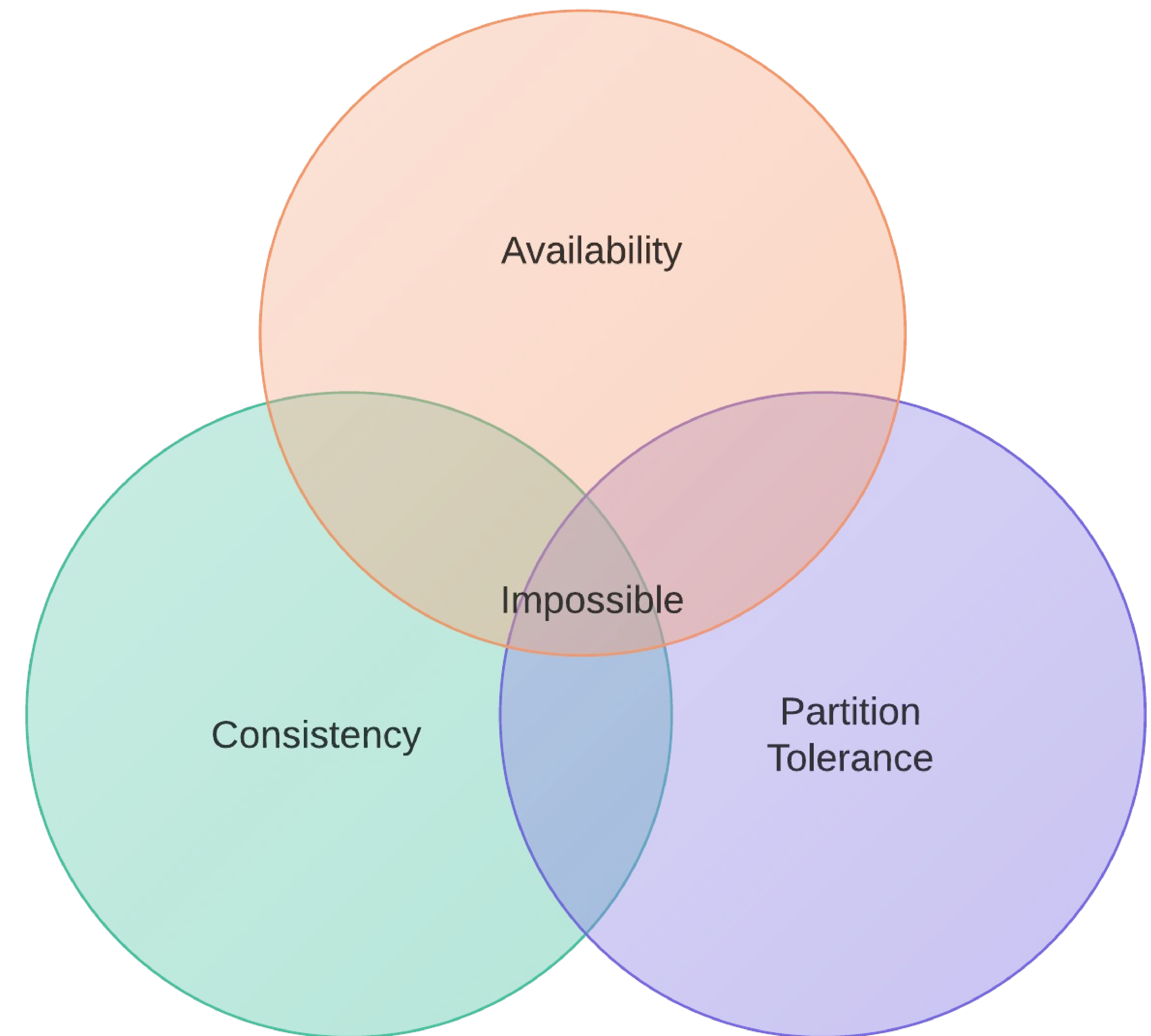
Sacrificing Consistency

“Just Like Dad”

softwarearchitecturecon.com
#oreillysacon

CAP Theorem

- Consistency – all nodes see the same data at the same time.
- Availability – all client requests will get a response.
- Partition Tolerance – the system continues despite message loss or partial system failure.

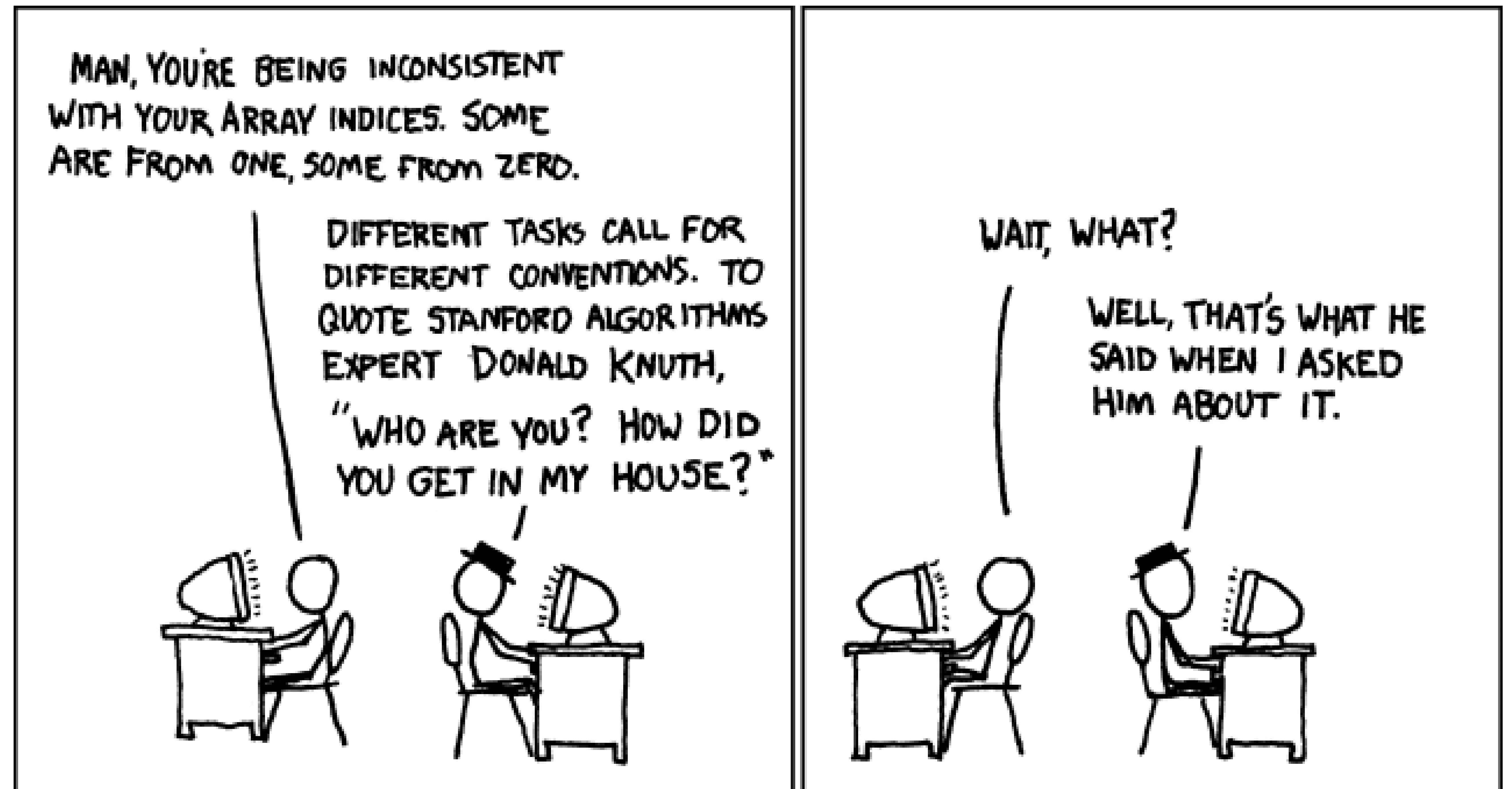


“Partitions are rare, there is little reason to forfeit consistency or availability when the system is not partitioned.... the choice between consistency and availability can occur many times within the same system at very fine granularity”

-Eric Brewer

Effective Cache Locations

- HTTP cache on client (memory and/or disk)
  - Hit rate scales poorly
  - Fast responses
- HTTP cache between client and server
  - Another service that can cascade failure
  - Prevents access to origin during failure
  - High hit rate
- Well-known cache off to the side
  - Another service that can fail
  - High hit rate



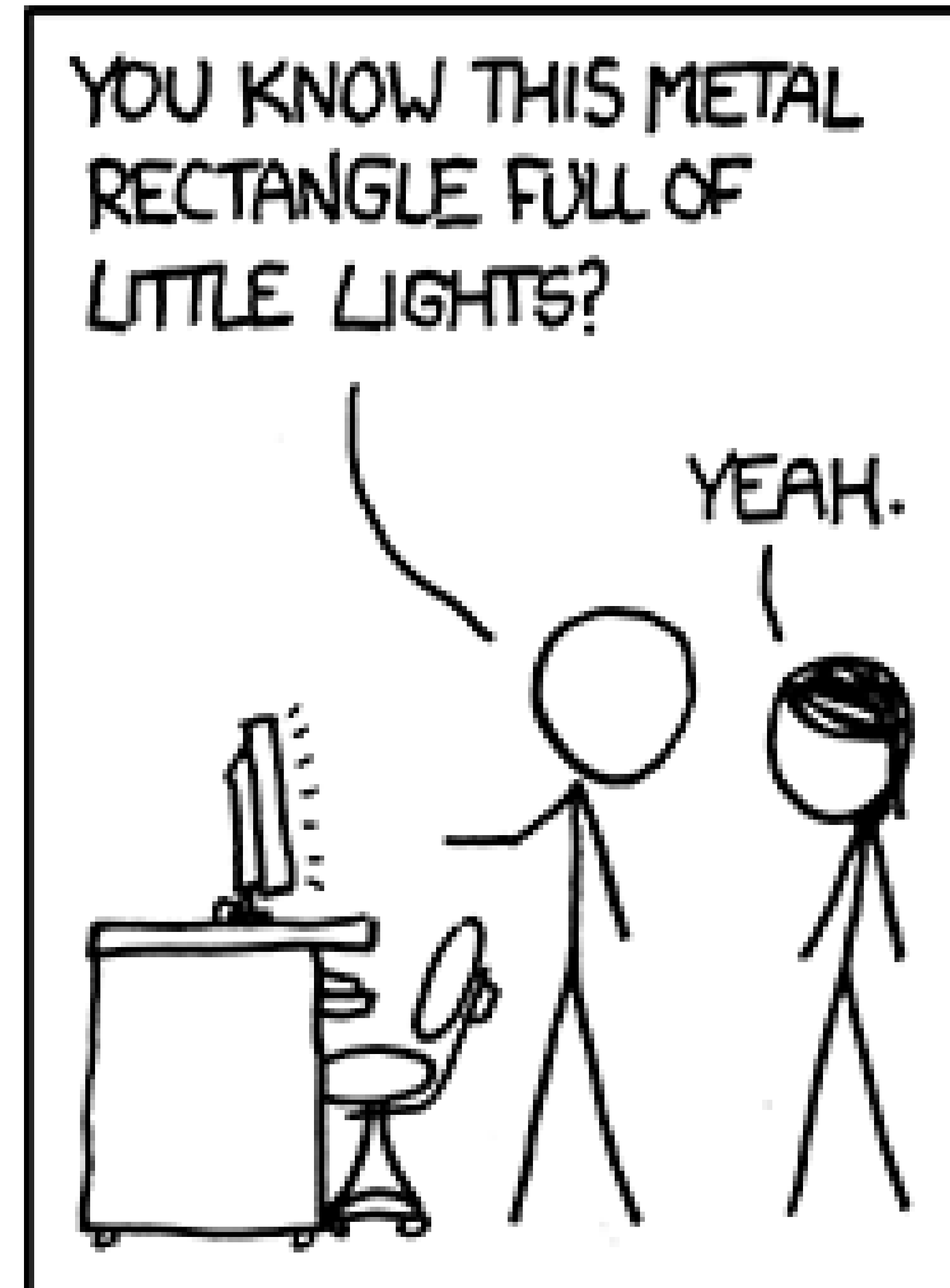
Effective Caching Strategies

- Respect caching headers from server
  - Allows server to determine consistency
  - Mitigates fewer failures
- Prefer origin, fallback to respected cache
  - Always consistent
  - Sometimes unavailable
- Store responses until overwritten, prefer origin
  - Always consistent unless availability would be sacrificed
  - Sometimes unavailable

Sometimes, the origin is down and the cache is empty.
You will need a failsafe.

Degraded Mode

- Show default options
- Give temporary access
- Lock out features
- Create a new account
- Assume the best
- Assume the worst
- It's your application, you decide!



O'REILLY®

Software Architecture

ENGINEERING THE FUTURE OF SOFTWARE

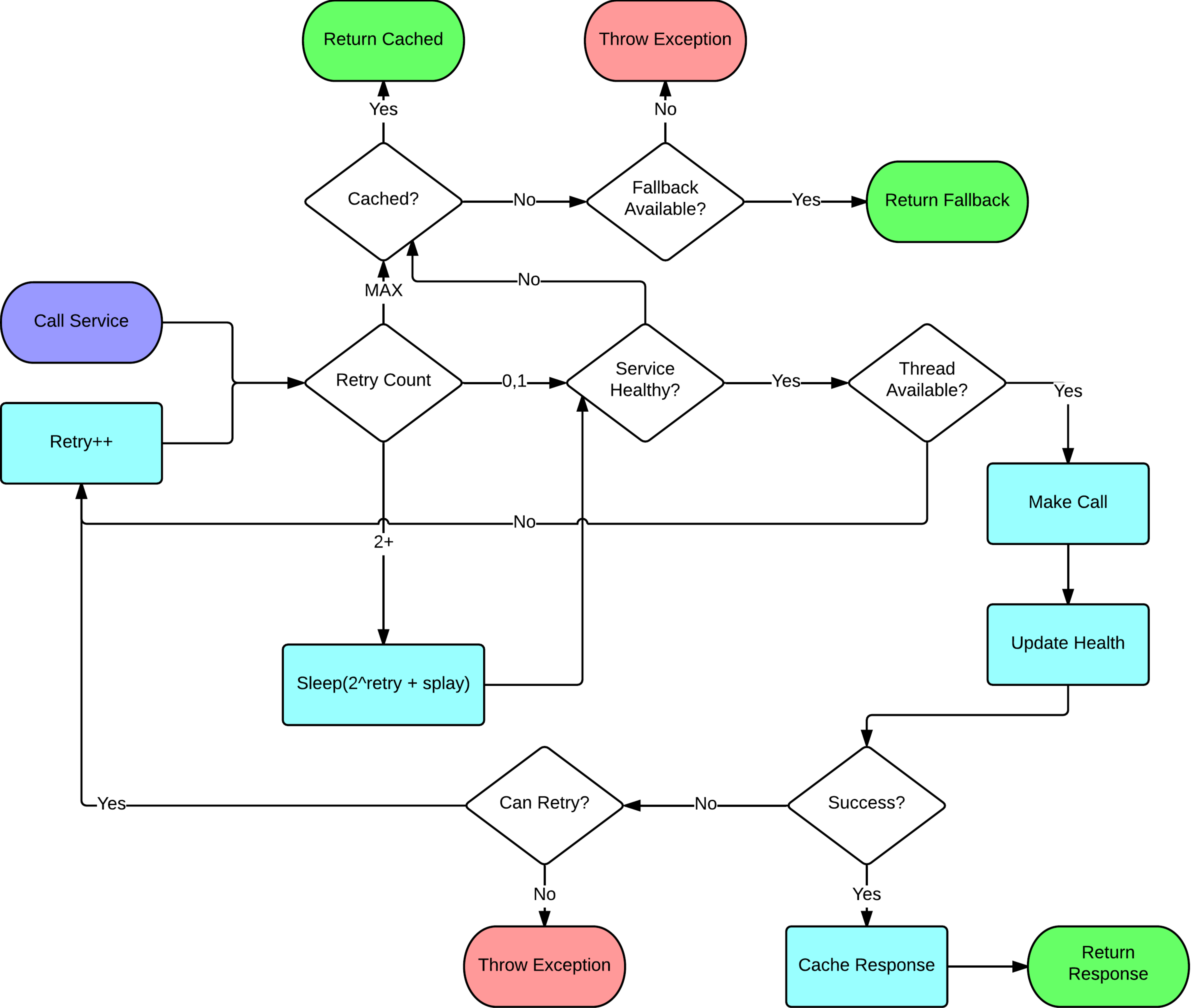
Lucid Software

On Availability

softwarearchitecturecon.com

#oreillysacon

Lucid Software's Service Calls



O'REILLY®

Software Architecture

ENGINEERING THE FUTURE OF SOFTWARE

Questions

Survey @ http://goo.gl/VDmCrt