

Advanced Systems Lab

G. Alonso, D. Kossmann, T. Roscoe

Systems Group

<http://www.systems.ethz.ch>

What you have learnt so far

- Methodology of Experiments
 - Ask questions, make hypothesis, ...
 - Real system vs. Model vs. Simulator
- You saw and created your first graphs
 - Brief discussion about throughput vs. resp. Time
- Workloads and Benchmarks
 - SPECint, TPC family, ...
- Monitoring
 - How to measure metrics?

Objectives of this lecture

- Understanding performance problems
 - bottlenecks
 - performance bugs
- Fixing performance problems
 - partitioning and replication
 - forms of parallelism
 - (using parallel databases as a vehicle)

Why are response times long?

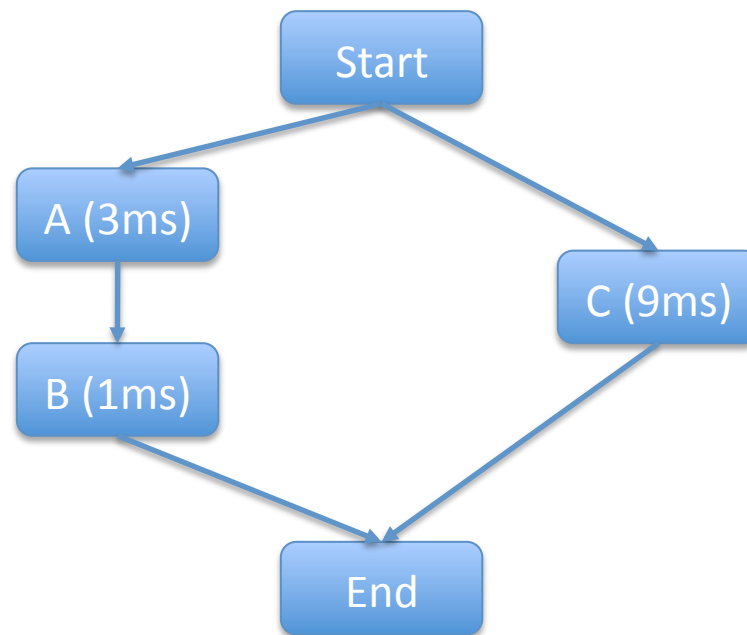
- Because operations take long
 - cannot travel faster than light
 - delays even in „single-user“ mode
 - possibly, „parallelize“ long-running operations
 - „intra-request parallelism“
- Because there is a bottleneck
 - contention of concurrent requests on a resource
 - requests wait in queue before resource available
 - add resources to parallelize requests at bottleneck
 - „inter-request parallelism“

Understanding Performance

- **Response Time**
 - critical path analysis in a task dependency graph
 - „partition“ expensive tasks into smaller tasks
- **Throughput**
 - queueing network model analysis (Weeks 8 & 9)
 - „replicate“ resources at bottleneck

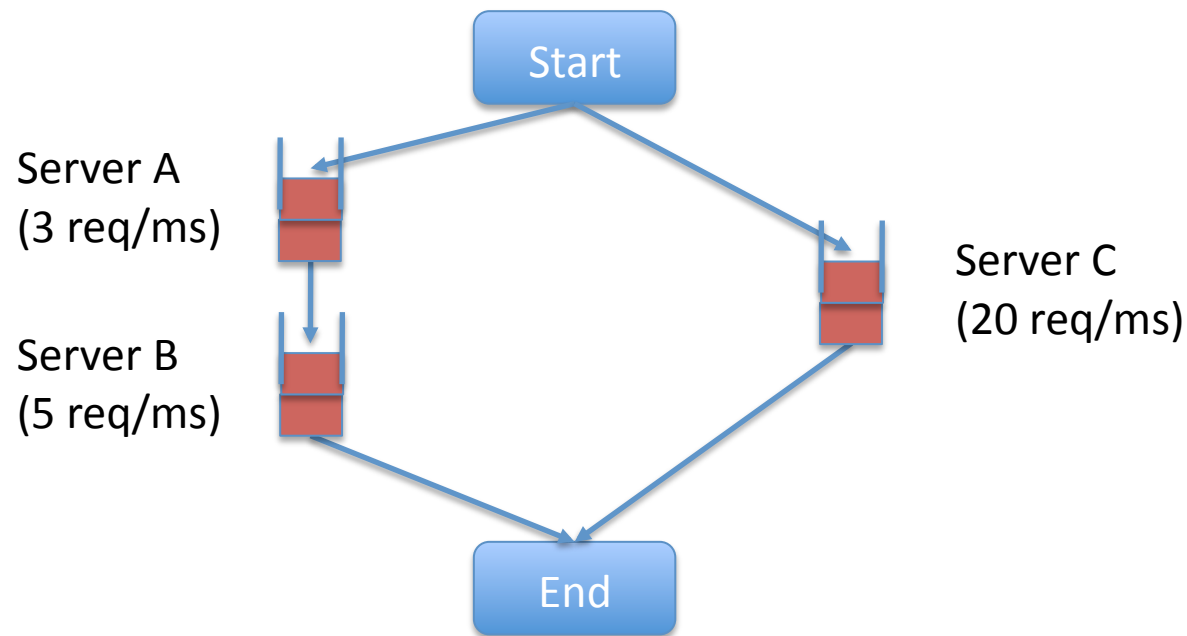
Critical Path

- Directed Graph of Tasks and Dependencies
 - response time = $\max \{ \text{length of path} \}$
 - assumptions: no resource contention, no pipelining, ...
- Which tasks would you try to optimize here?



Queueing Network Models

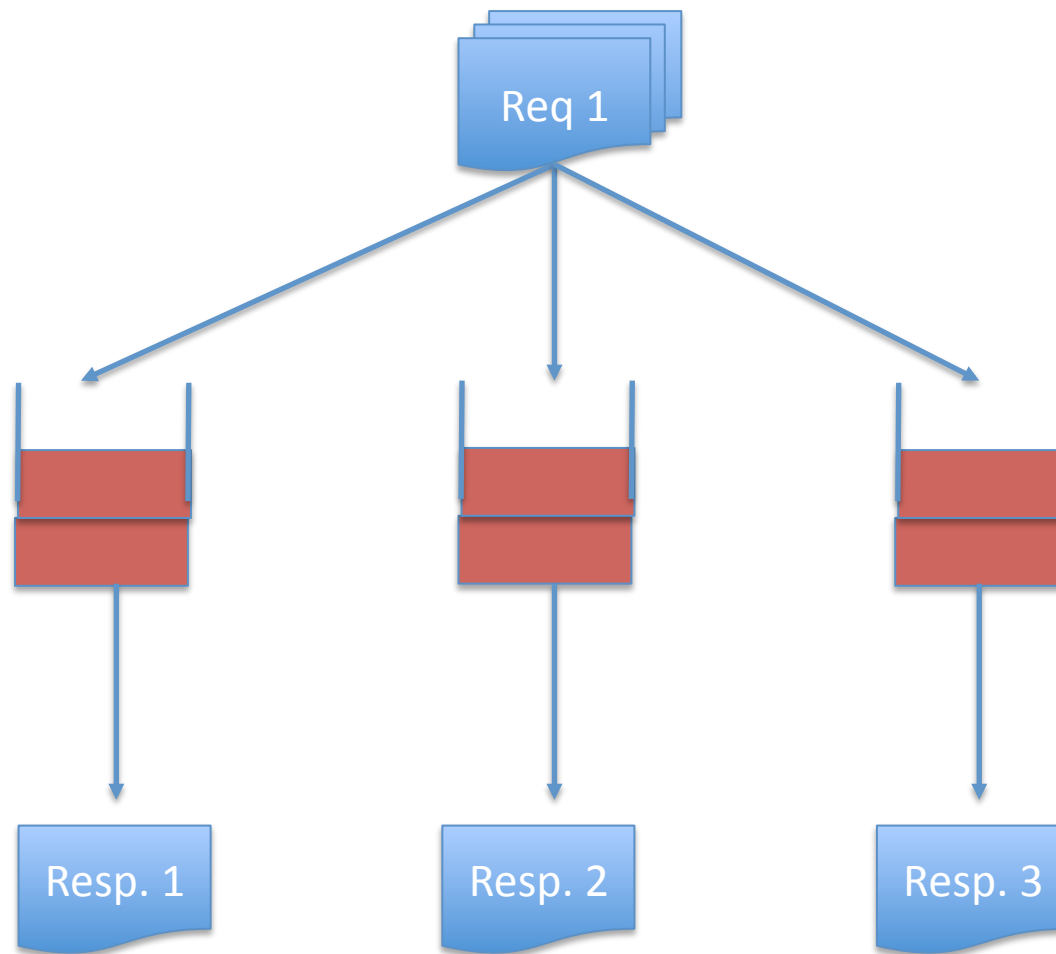
- Graph of resources and flow of requests
- Bottleneck=Resource defines tput of whole system
 - (analysis techniques described in Weeks 8 & 9)



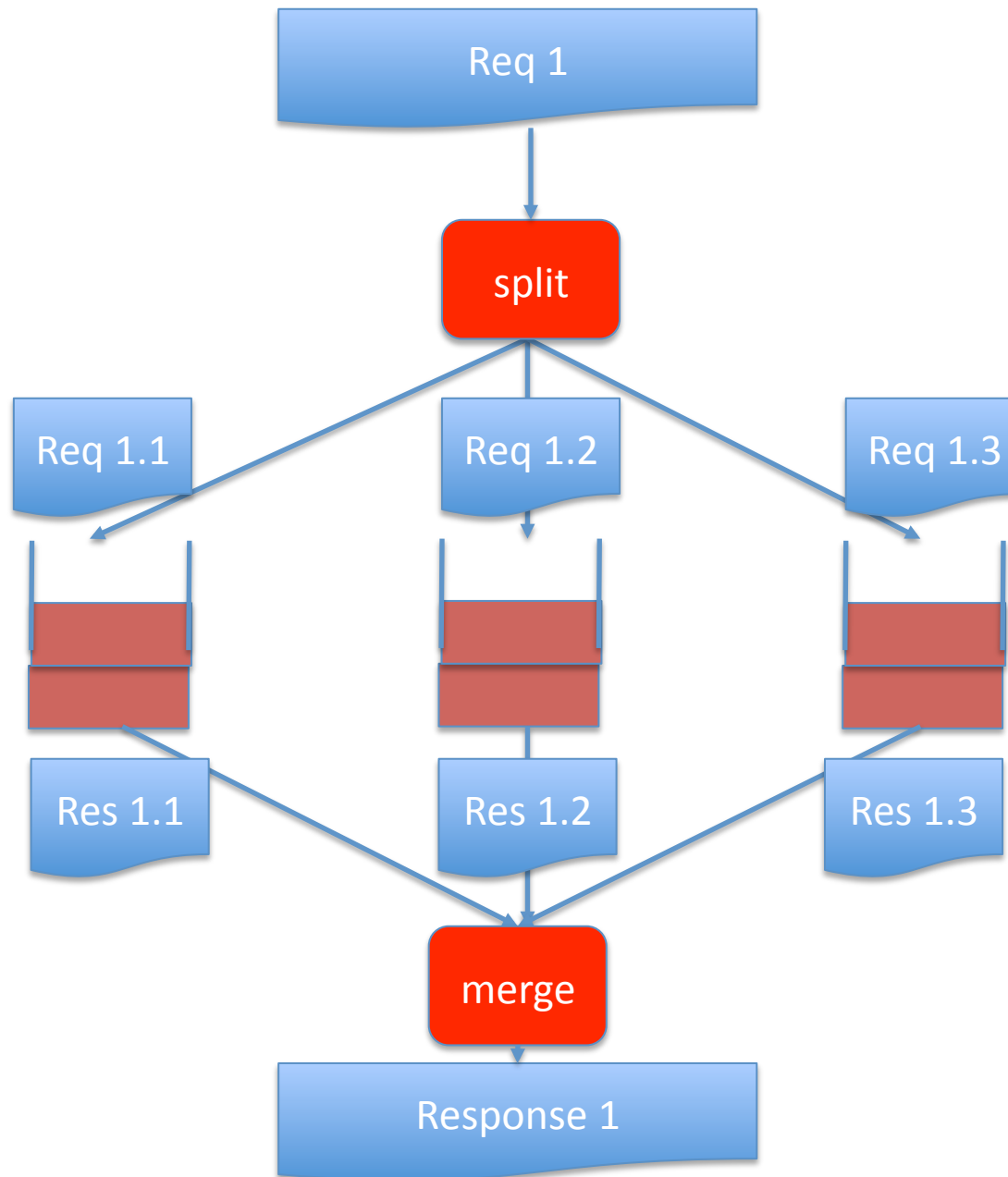
Forms of Parallelism

- **Inter-request Parallelism**
 - several requests handled at the same time
 - principle: replicate resources
 - e.g., ATMs
- **(Independent) Intra-request Parallelism**
 - principle: divide & conquer
 - e.g., print pieces of document on several printers
- **Pipelining**
 - each „item“ is processed by several resources
 - process „items“ at different resources in parallel
 - can lead to both inter- & intra-request parallelism

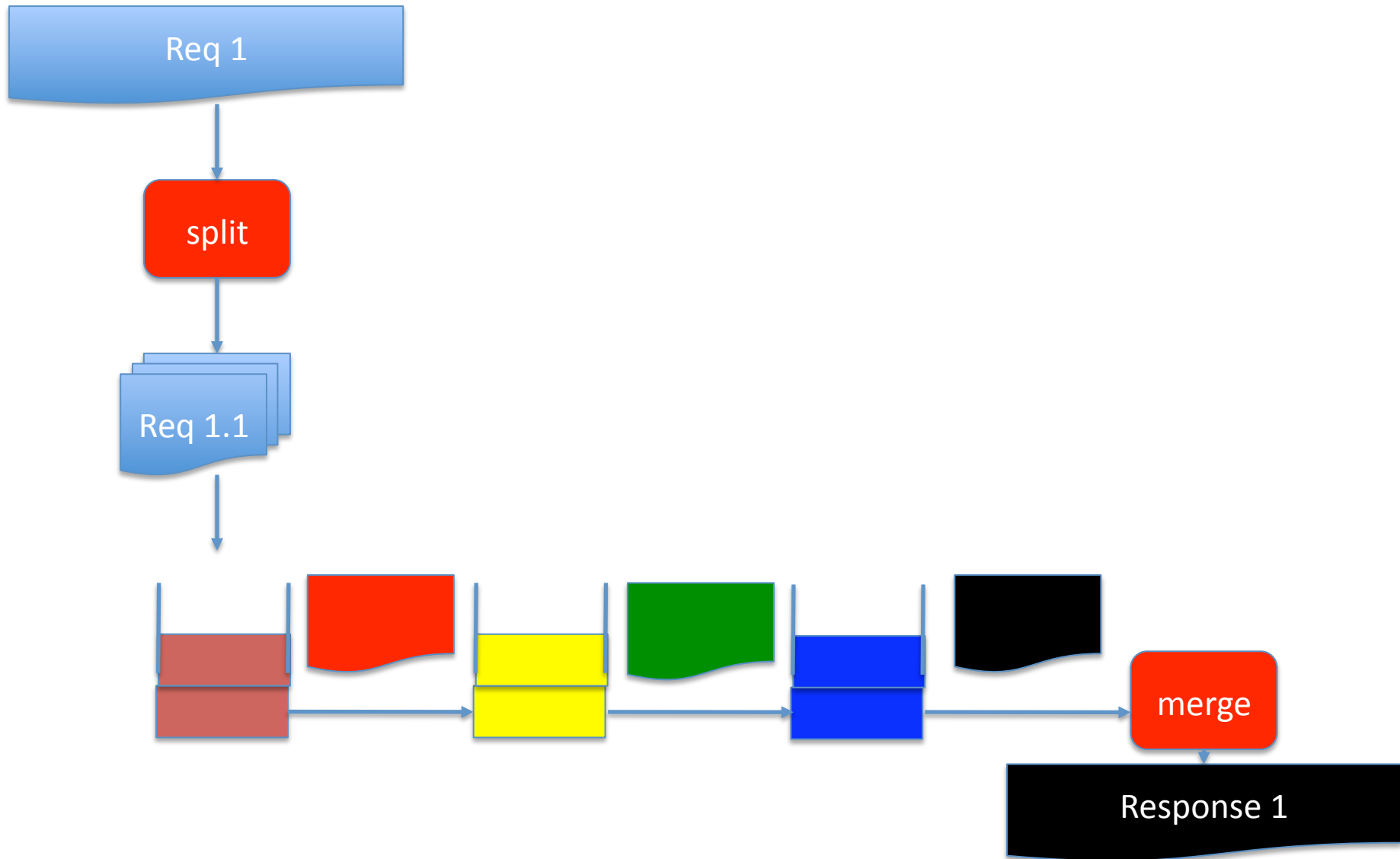
Inter-request Parallelism



Independent Parallelism



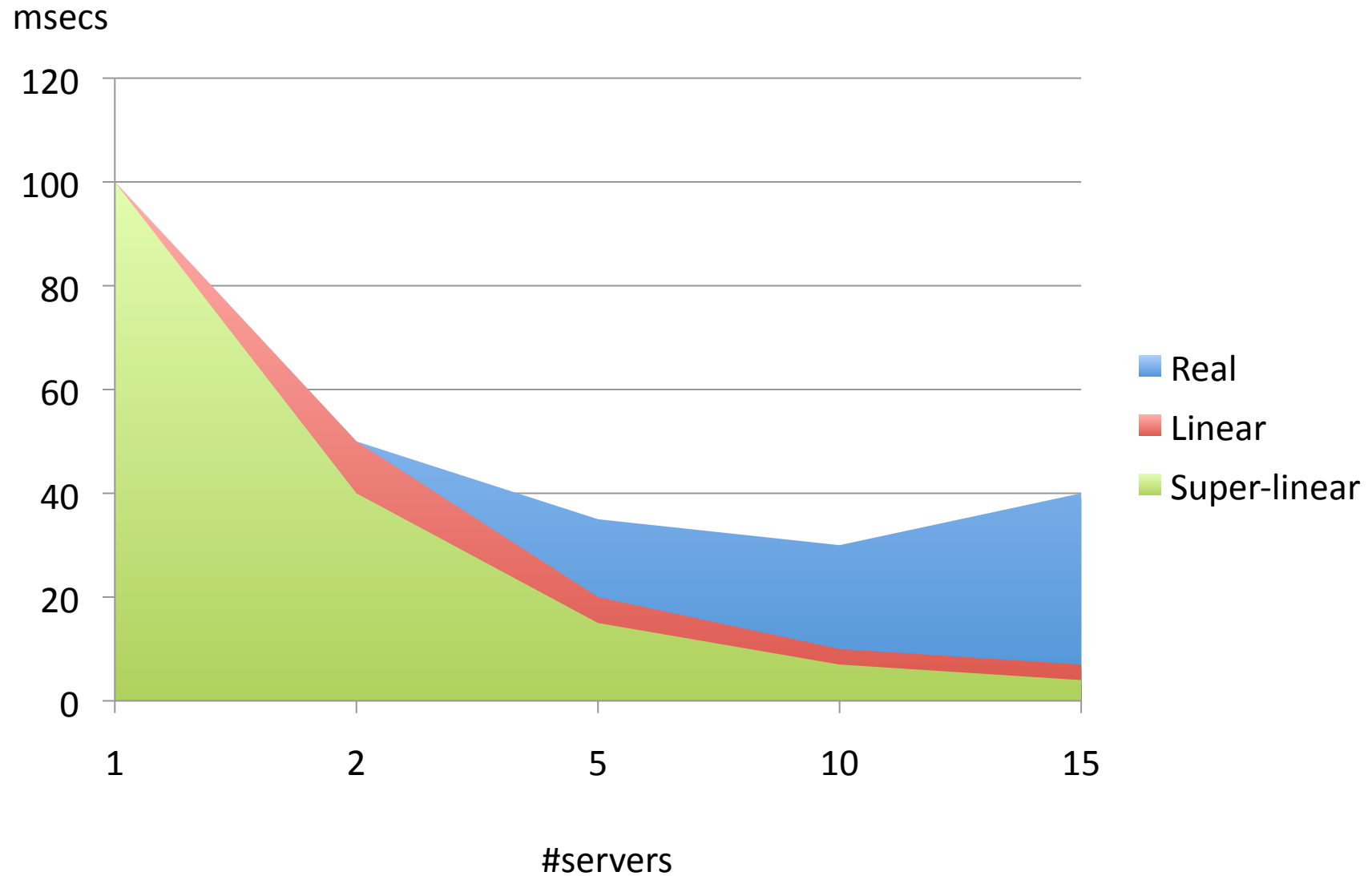
Pipelining (Intra-request)



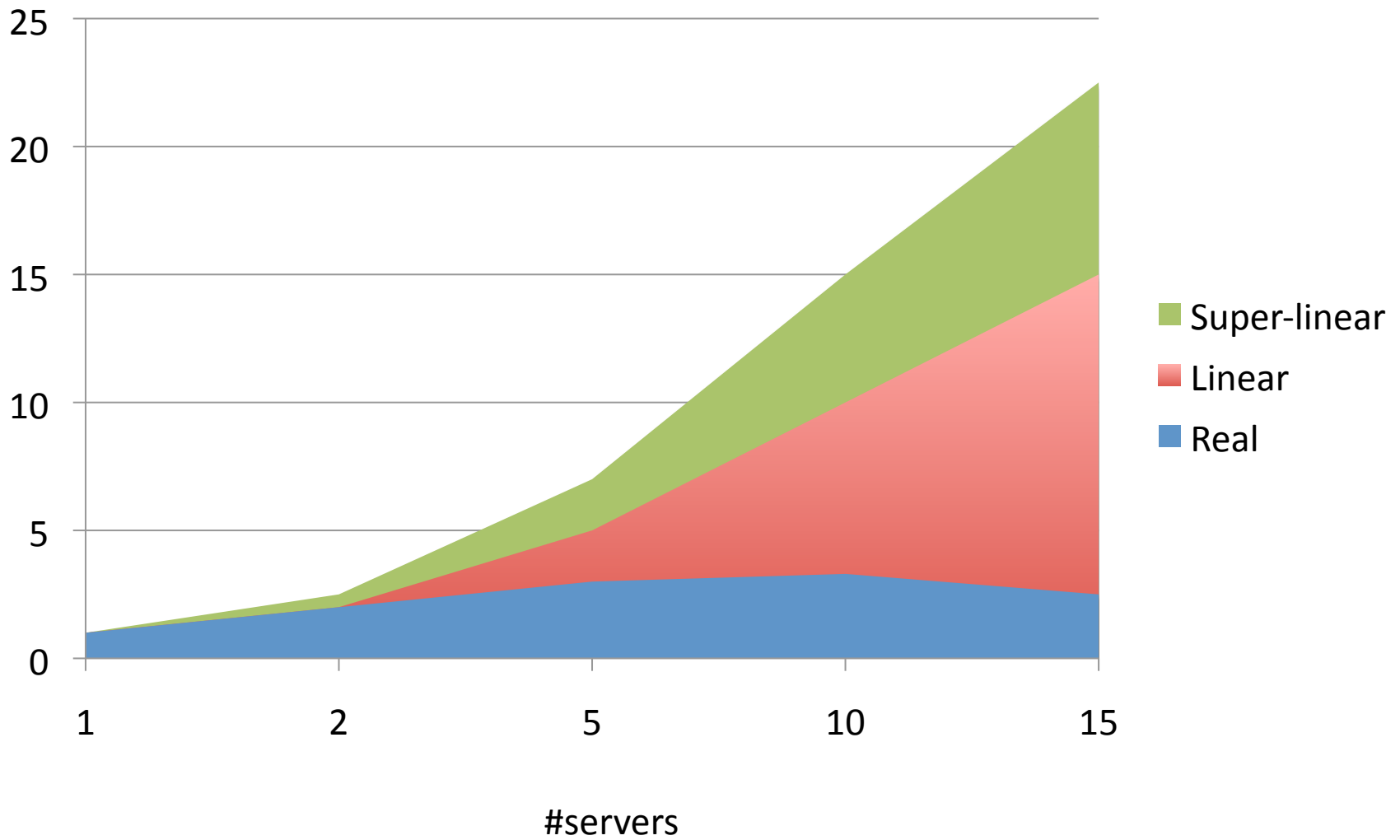
Speed-up

- Metric for intra-request parallelization
- Goal: test ability of SUT to reduce response time
 - measure response time with 1 resource
 - measure response time with N resources
 - $\text{SpeedUp}(N) = \text{RT}(1) / \text{RT}(N)$
- Ideal
 - $\text{SpeedUp}(N)$ is a linear function
 - can you imagine super-linear speed-ups?

Response Times



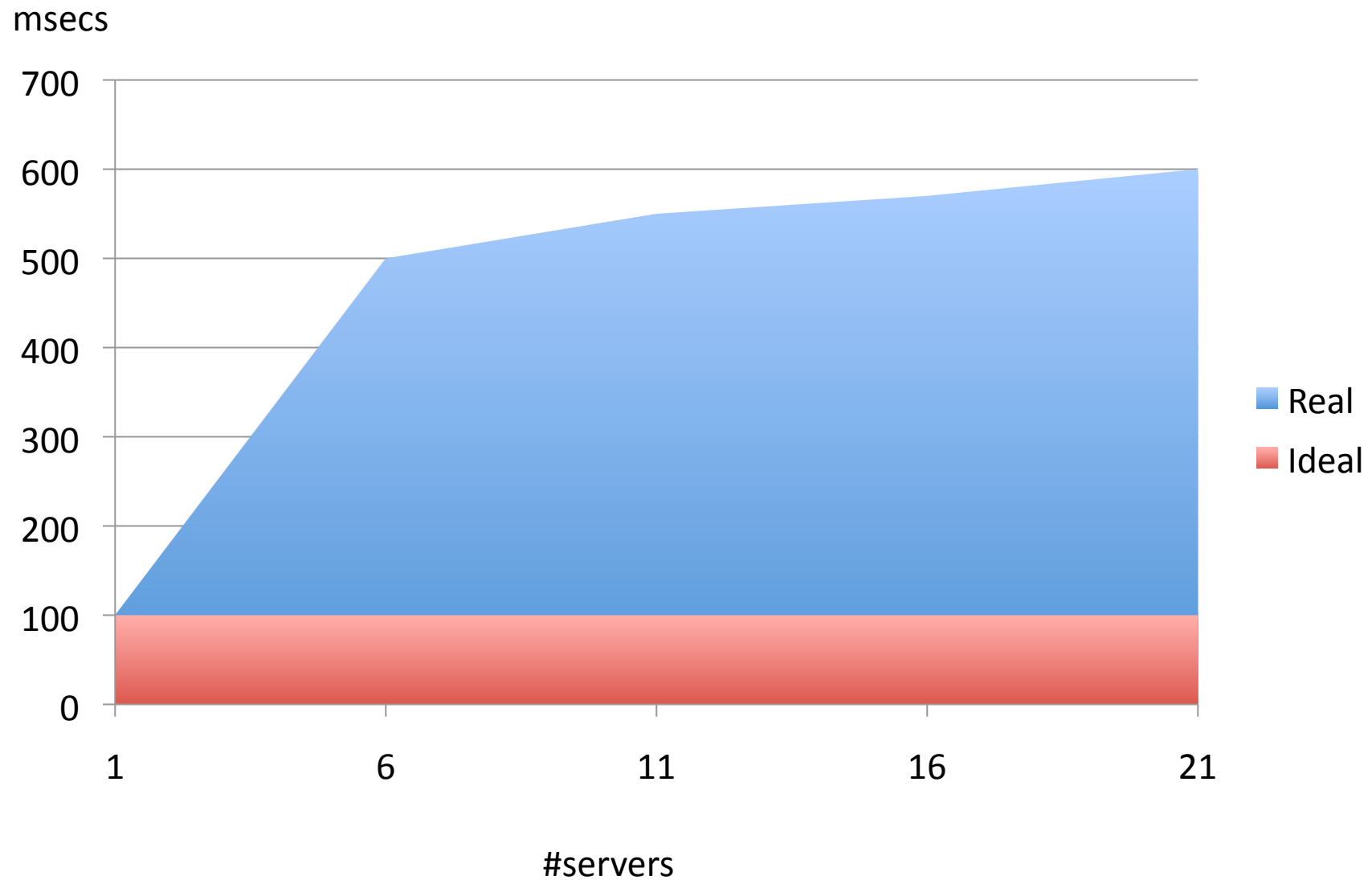
Speed Up



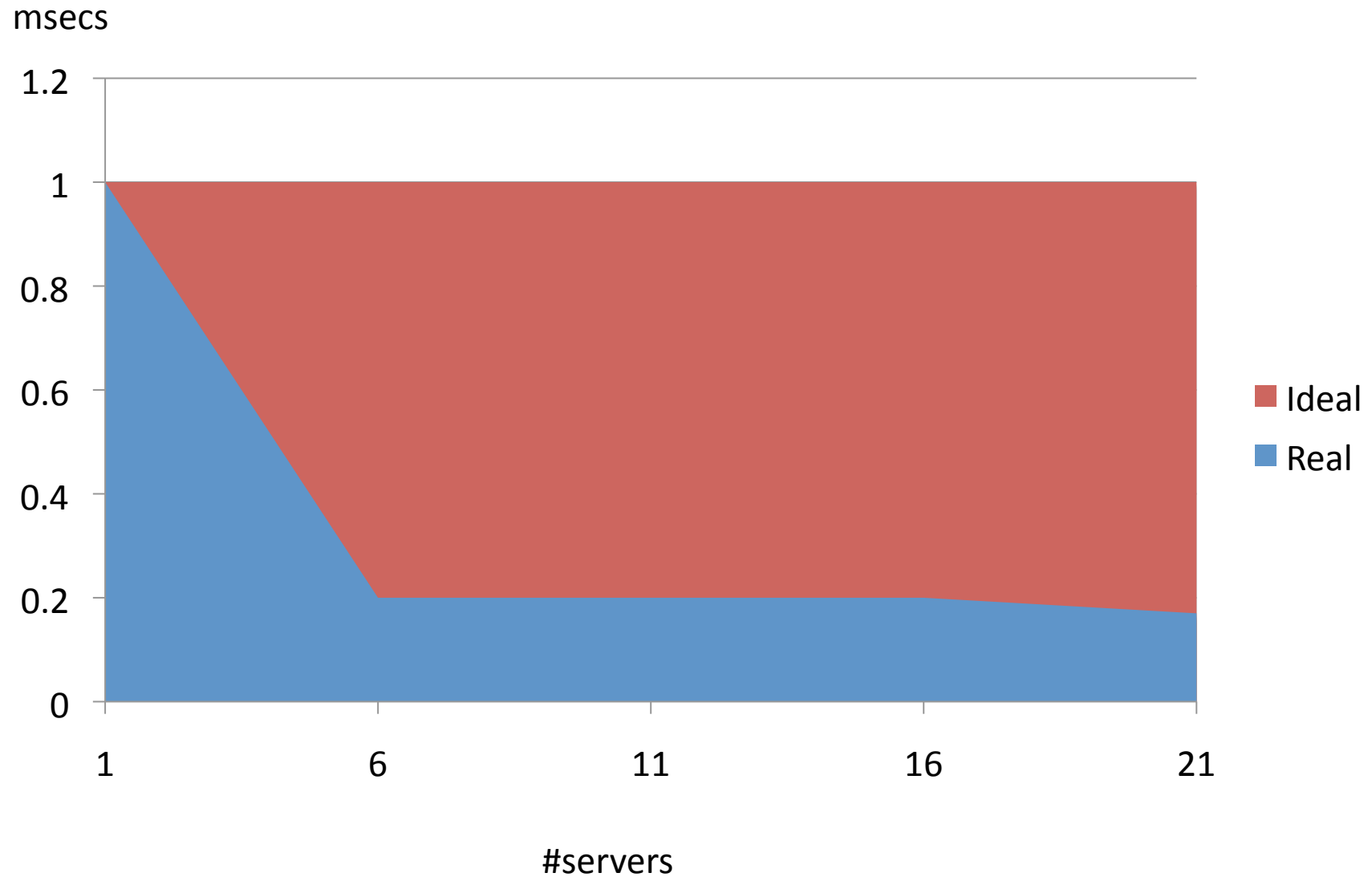
Scale-up

- Test how SUT scales with size of the problem
 - measure response time with 1 server, unit problem
 - measure response time with N servers, N units problem
 - $\text{ScaleUp}(N) = \text{RT}(1) / \text{RT}(N)$
- Ideal
 - $\text{ScaleUp}(N)$ is a constant function
 - Can you imagine super scale-up?

Scale Up Exp.: Response Time



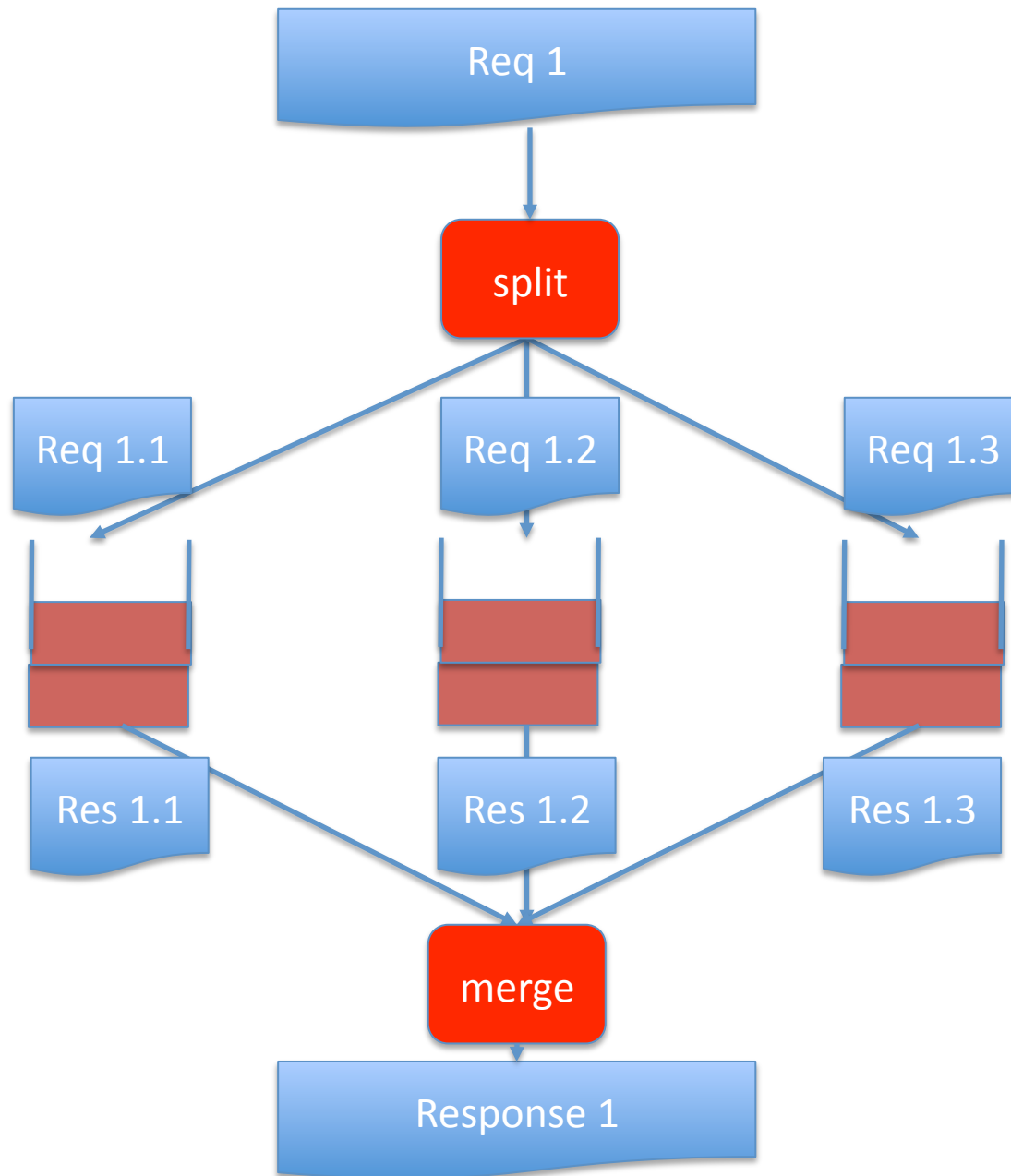
Scale Up Exp.: Response Time



Scale Out

- Test how SUT behaves with increasing load
 - measure throughput: 1 server, 1 user
 - measure throughput: N servers, N users
 - $\text{ScaleOut}(N) = \text{Tput}(1) / \text{Tput}(N)$
- Ideal
 - Scale-Out should behave like Scale-Up
 - (often terms are used interchangeably; but worth-while to notice the differences)
- Scale-out and down in Cloud Computing
 - the ability of a system to adapt to changes in load
 - often measured in \$ (or at least involving cost)

Why is speed-up sub-linear?



Why is speed-up sub-linear?

- **Cost for „split“ and „merge“ operation**
 - those can be expensive operations
 - try to parallelize them, too
- **Interference: servers need to synchronize**
 - e.g., CPUs access data from same disk at same time
 - shared-nothing architecture
- **Skew: work not „split“ into equal-sized chunks**
 - e.g., some pieces much bigger than others
 - keep statistics and plan better

How to split a problem?

- Cost model to split problem into „p“ pieces

$$\text{Cost}(p) = a * p + (b * K) / p$$

- a: constant overhead per piece for split & merge
- b: constant overhead per item of the problem
- K: total number of items in the problem

- Minimize this function

- simple calculus: $\text{Cost}(p)'=0$; $\text{Cost}(p)'' > 0$

$$p = \text{sqrt}(b * K / a)$$

Distributed & Parallel Databases

- **Distributed Databases (e.g., banks)**
 - partition the data
 - install database nodes at different locations
 - keep partitions at locations where frequently needed
 - if beneficial replicate partitions / cache data
 - goal: reduce communication cost
- **Parallel Databases (e.g., Google)**
 - partition the data
 - install database nodes within tightly-coupled network
 - goal: speed-up by parallel queries on partitions

Why are PDDBs so cool? ;-)

- Data is a „resource“ (just like a server)
 - data can be a bottleneck if it is updated
 - data can be replicated in order to improve tput
- Data is a „problem“
 - data can be partitioned in good and poor ways
 - partitioning can be done statically and dynamically
 - If statically, then „split“ operation is free
- Data can be used for scalability experiments
 - you can nicely show all
 - database systems are freely available (e.g., PostGRES)

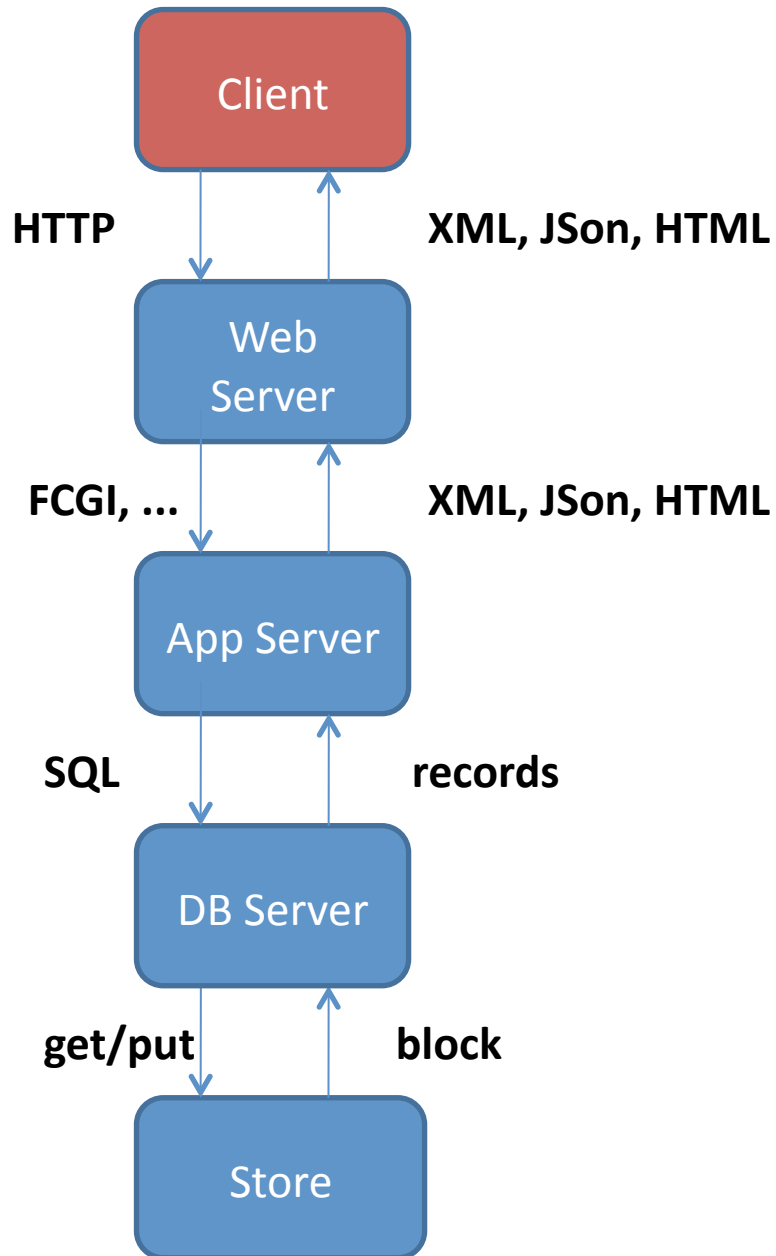
How to partition data? (M3 😊)

- (here: horizontal partitioning only)
- **Step 1: Need to determine partitioning factor**
 - very difficult task; depends on many factors
- **Step 2: Determine partitioning method**
 - Round-robin: good for load balancing
 - Predicate-based: good for certain queries (e.g., sort)
 - Hashing: good for „key“ look-ups and updates
 - ... choose your poison
- **Step 3: Determine allocation**
 - which partition to replicate and how often
 - where to store replicas of each partition

Replication

- **Dimension 1: How to keep replicas consistent**
 - ROWA: read-one, write all
 - Quorum-based approaches
 - Master copy
- **Dimension 2: Kinds of replication**
 - mirroring: each DB node has exactly one replica
 - when a node dies, its replica takes over
 - scattered: replicas of one DB node distributed in group
 - when a node dies, a set of nodes take over its duties
 - Each node is primary server for some part., replica for others
 - chained: like scattered, but clear rule of distribution
 - example: DHTs

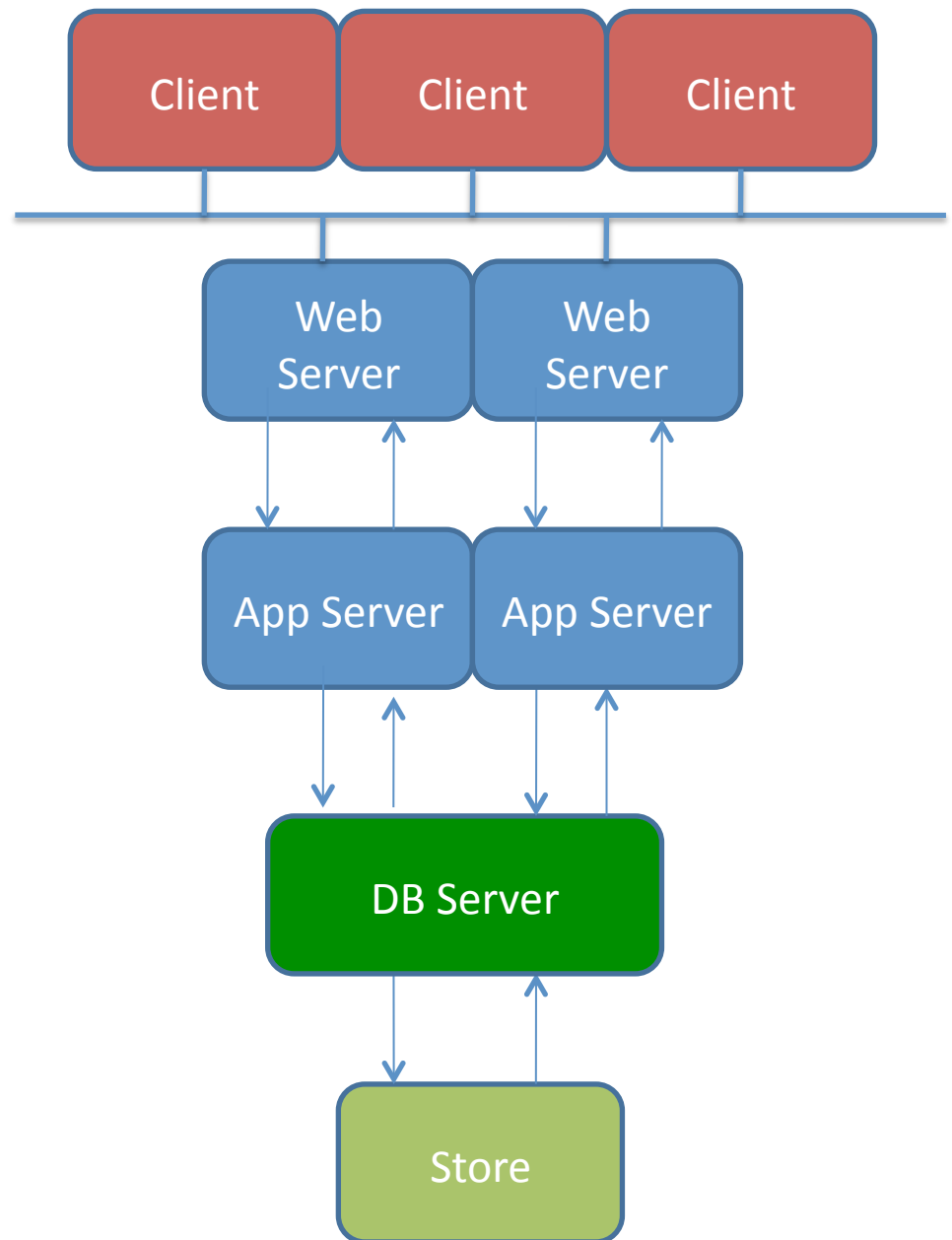
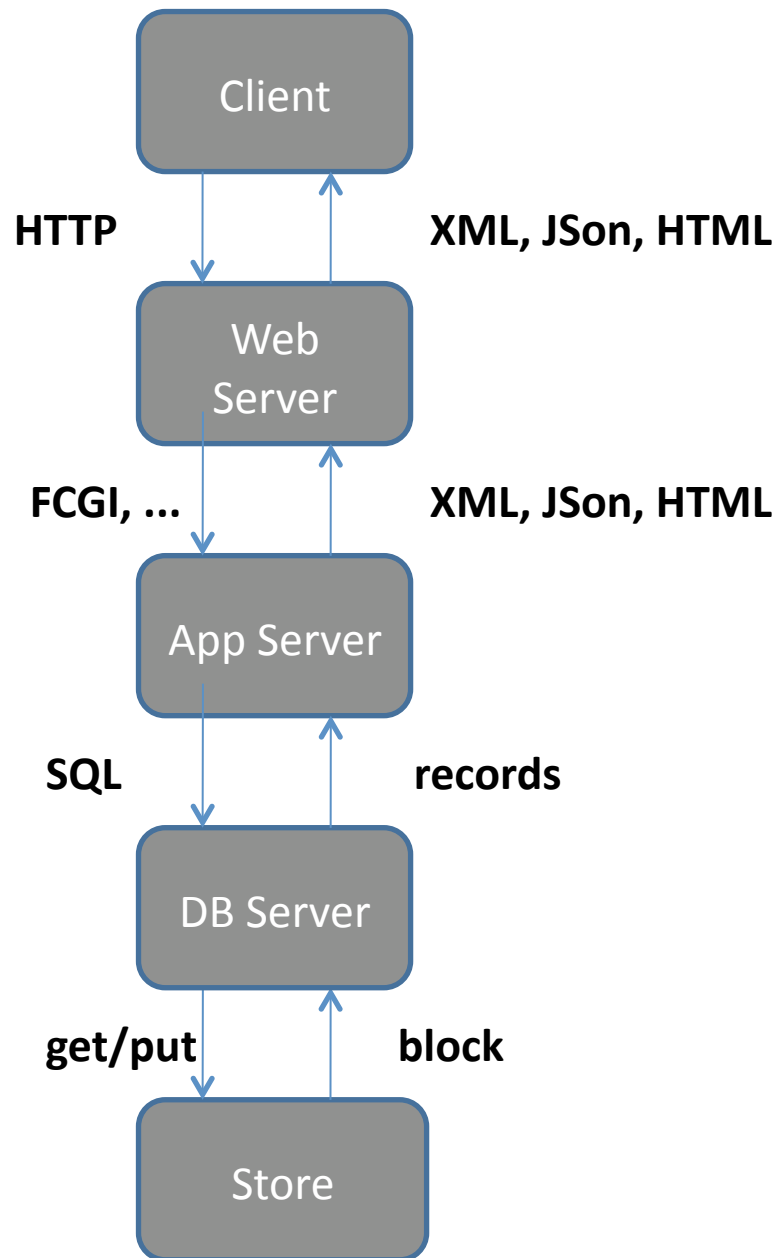
Reference Database Architecture



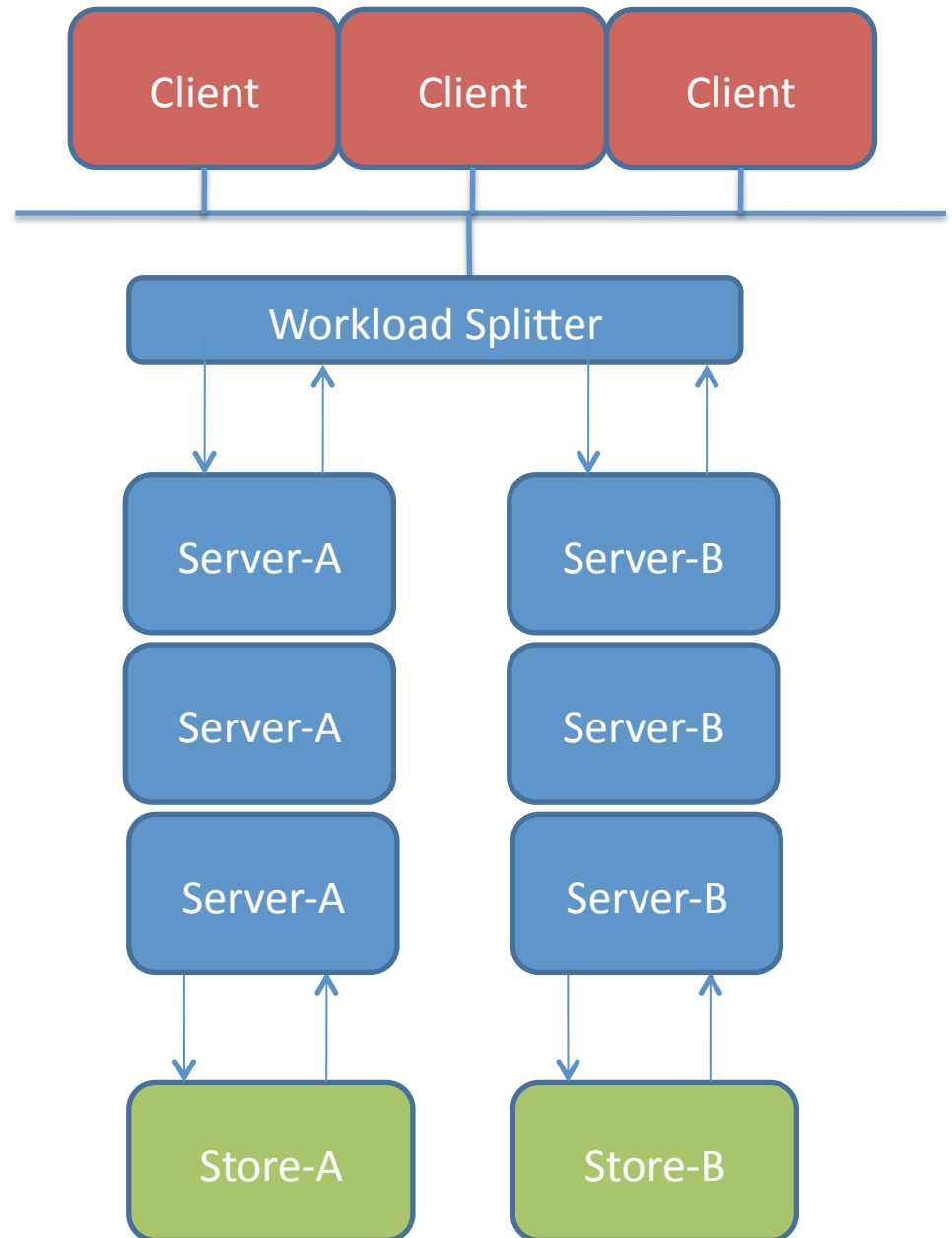
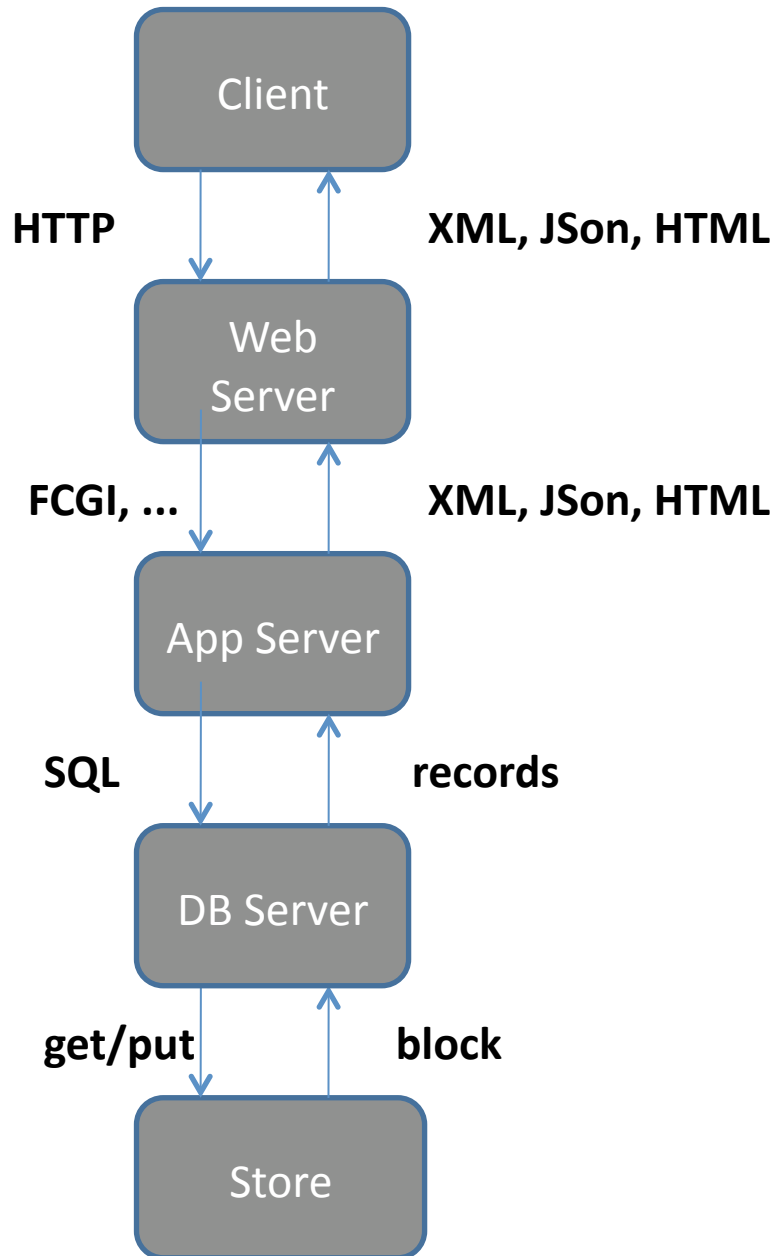
Typical Bottlenecks

- Storage System (solved)
 - Increase bandwidth with disk arrays
- Database Server (unsolved)
 - Use big machines (traditional) or
 - DHTs (cloud)
- Application Server, Web Server (solved)
 - Add machines at this tier
 - Keep app servers stateless

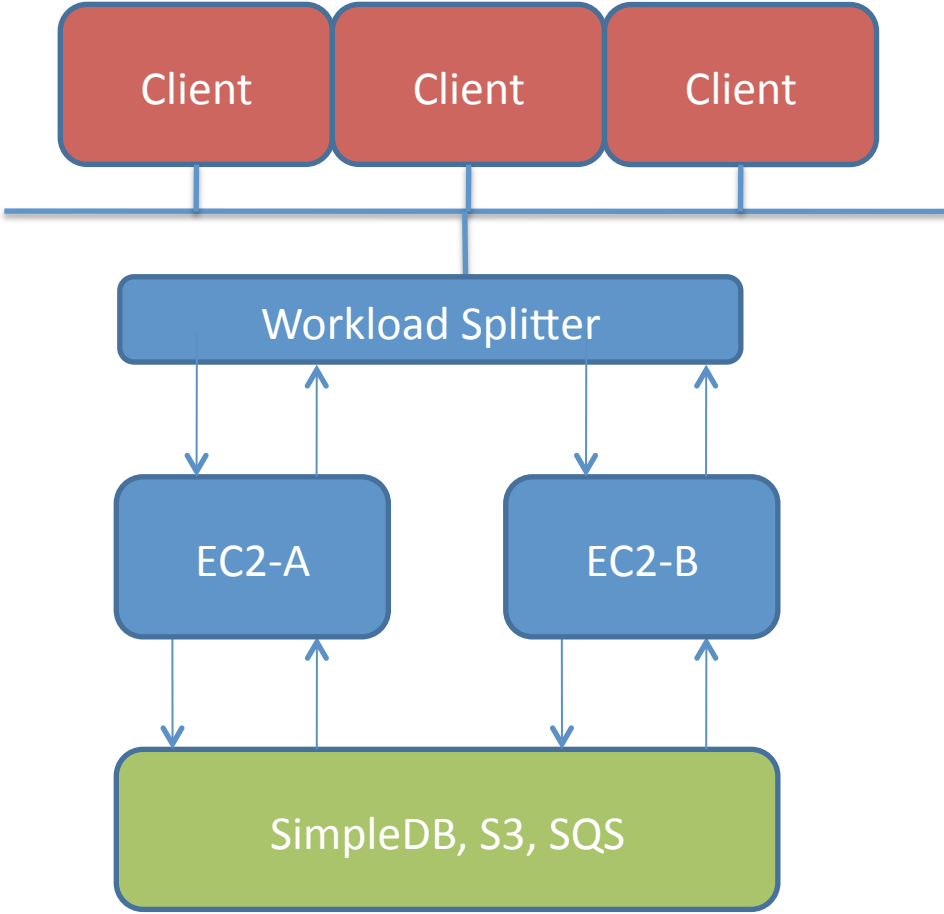
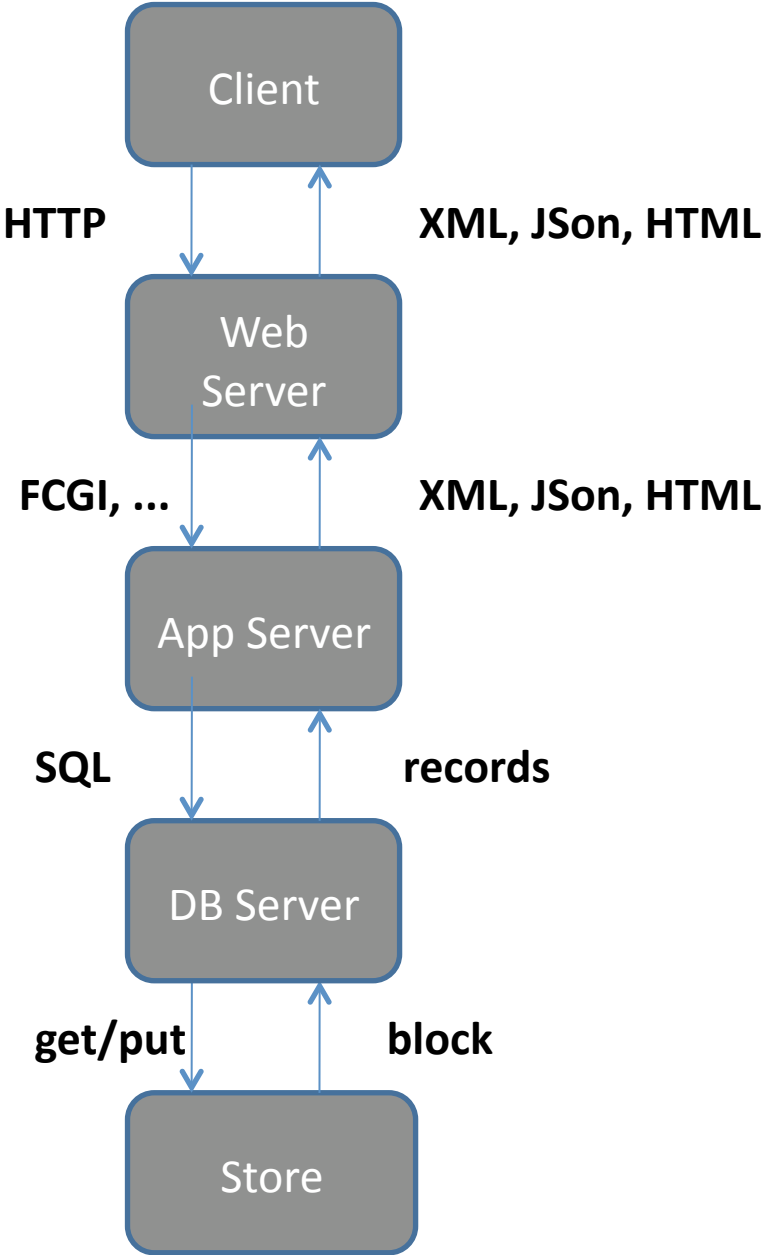
Traditional Deployment (e.g., SAP R3)



Force.com Architecture



Cloud Architecture (e.g., Google AE)

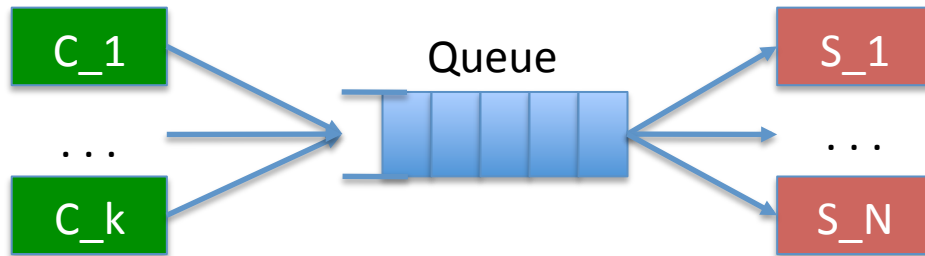


Example Performance Bugs

- **Unnecessary queues**
 - Sometimes direct communication better
 - Queues only needed if there is variance in speed
- **Unnecessary synchronization**
 - Results in locking overhead and blocking of computation
- **Copying of data**
 - Move pointers not tuples
 - Avoid „malloc“ and „free“ in the inner loop

What is problematic here?

- k clients, N servers and 1 queue of requests



- What is wrong with the following implementation?

```
int s = 0;
while(true) {
    msg = dequeue(input);
    result = server[s](msg);
    enqueue(output, result);
    s = (s+1) mod N;
}
```

Summary

- Improve Response Times by „partitioning“
 - divide & conquer approach
 - works extremely well for databases and SQL
- Improve Throughput by relaxing „bottleneck“
 - add resources at bottleneck
- Fundamental limitations to scalability
 - resource contention (e.g., lock conflicts in DB)
 - skew and poor load balancing
- Special kinds of experiments for scalability
 - speed-up and scale-up experiments