Advanced Distributed Systems

RPCs & MapReduce

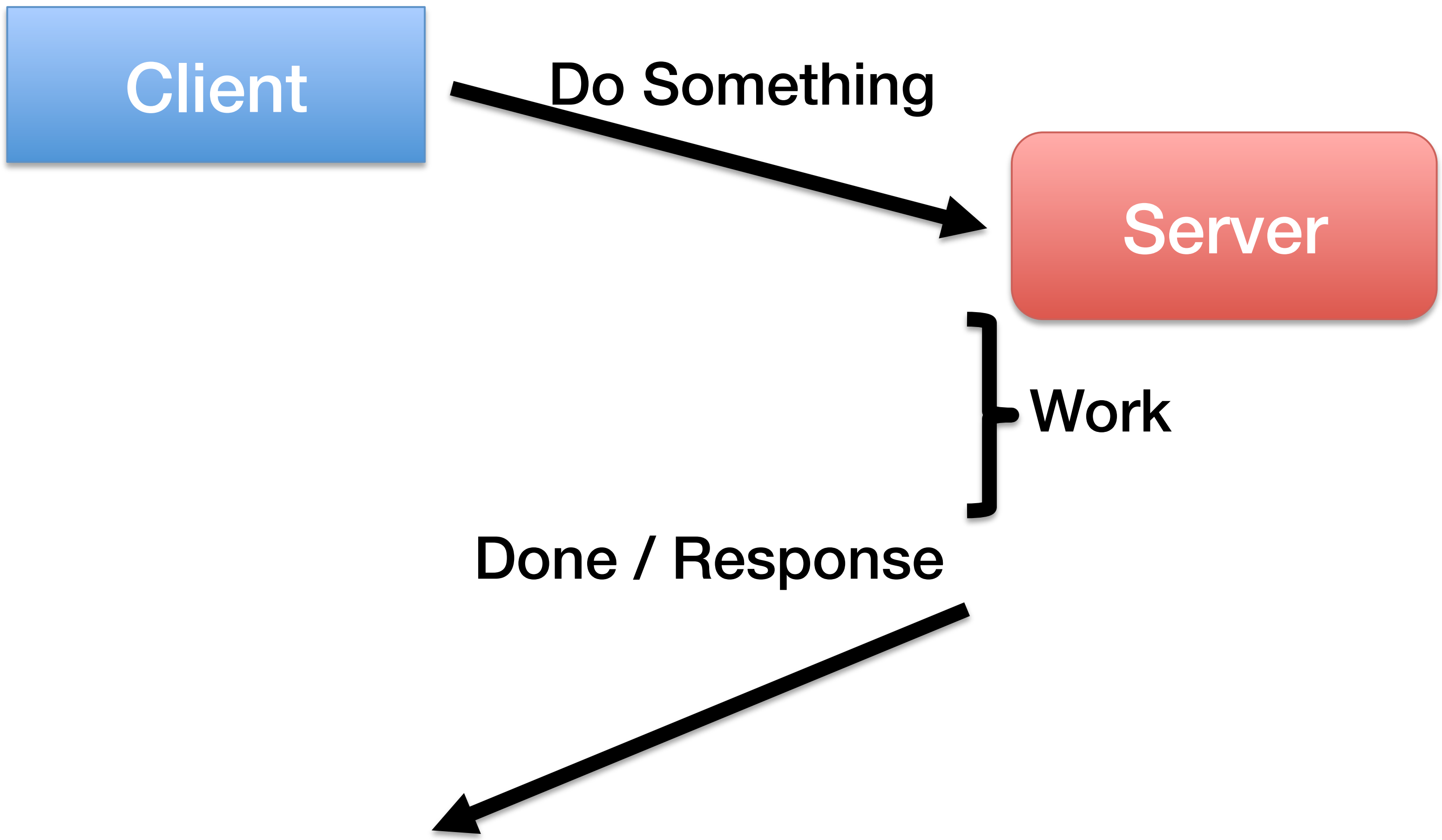
Wyatt Lloyd

Some slides adapted from:
Dave Andersen/Srini Seshan;
Lorenzo Alisi/Mike Dahlin;
Frans Kaashoek/Robert Morris/Nickolai Zeldovich;
Jinyang Li;
Jeff Dean;

Remote Procedure Call (RPC)

- Key question:
  - "What programming abstractions work well to split work among multiple networked computers?"

Common Communication Pattern



Alternative: Sockets

- Manually format
- Send network packets directly

```
struct foomsg {
  u_int32_t len;
}

send_foo(char *contents) {
  int msglen = sizeof(struct foomsg) + strlen(contents);
  char buf = malloc(msglen);
  struct foomsg *fm = (struct foomsg *)buf;
  fm->len = htonl(strlen(contents));
  memcpy(buf + sizeof(struct foomsg),
         contents,
         strlen(contents));
  write(outsock, buf, msglen);
}
```

Remote Procedure Call (RPC)

- Key piece of distributed systems machinery
- Goal: easy-to-program network communication
  - hides most details of client/server communication
  - client call is much like ordinary procedure call
  - server handlers are much like ordinary procedures
- RPC is widely used!
  - Google: Protobufs
  - Facebook: Thrift
  - Twitter: Finalge

RPC Example

- RPC ideally makes network communication look just like a function call
- Client:
  ```
  z = fn(x, y)
  ```
- Server:
  ```
  fn(x, y) {
    compute
    return z
  }
  ```
- RPC aims for this level of transparency
- Hope: even novice programmers can use function calls!

RPC since 1983

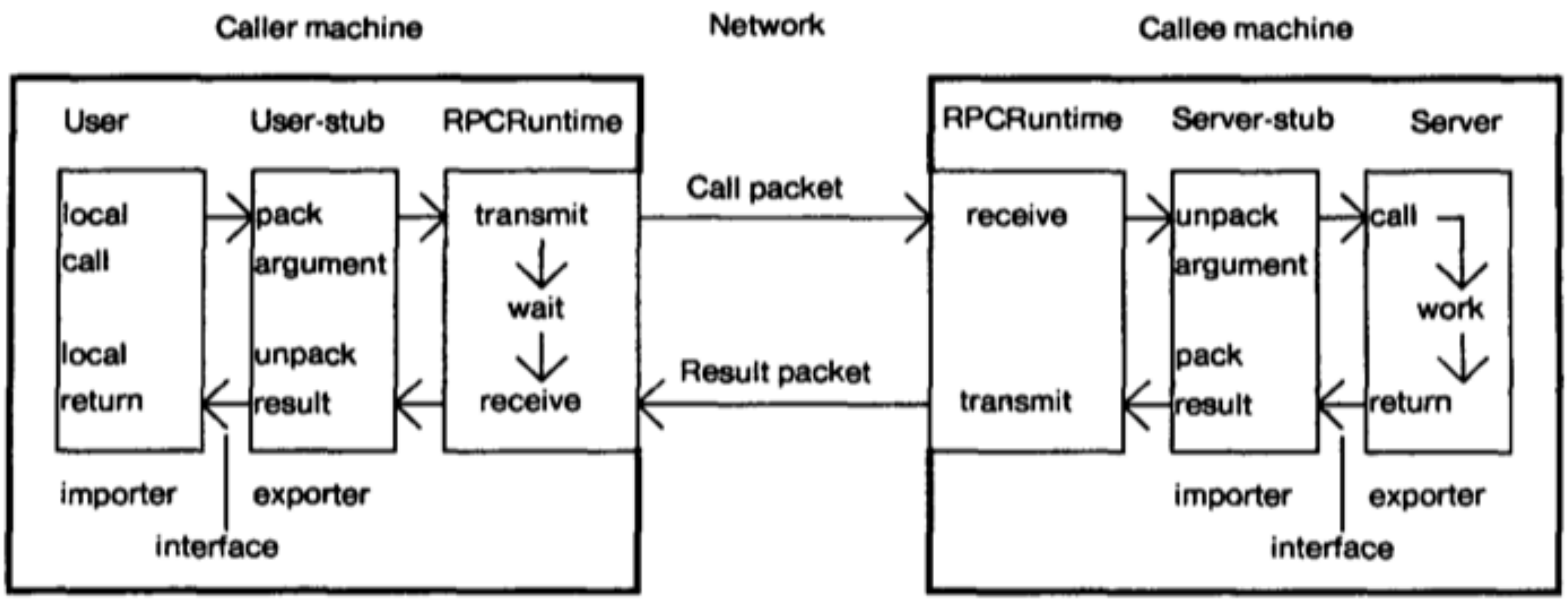


Fig. 1. The components of the system, and their interactions for a simple call.

RPC since 1983

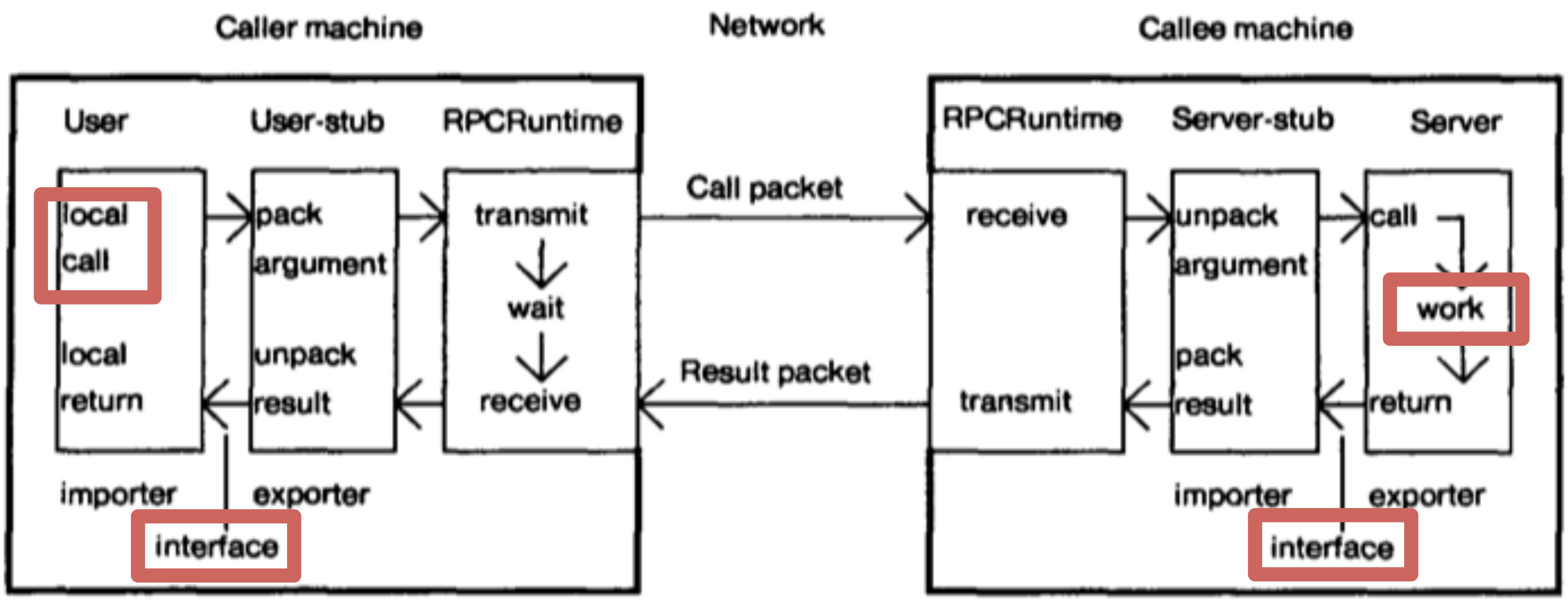


Fig. 1. The components of the system, and their interactions for a simple call.

What the programmer writes.

RPC Interface

- Uses interface definition language

```
service MultiplicationService
{
	int multiply(int n1, int n2),
}
```

```
MultigetSliceResult multiget_slice(1:required list<binary> keys,
            2:required ColumnParent column_parent,
            3:required SlicePredicate predicate,
            4:required ConsistencyLevel consistency_level=ConsistencyLevel.ONE,
            99: LamportTimestamp lts)
    throws (1:InvalidRequestException ire, 2:UnavailableException ue,
3:TimedOutException te),
```

RPC Stubs

- Generates boilerplate in specified language
  - (Level of boilerplate varies, Thrift will generate servers in C++, …

```
$ thrift --gen go multiplication.thrift
```

- Programmer needs to setup connection and call generated function

```
client = MultiplicationService.Client(…)
client.multiply(4.5)
```

- Programmer implements server side code

```
public class MultiplicationHandler implements MultiplicationService.Iface {

public int multiply(int n1, int n2) throws TException {
       System.out.println("Multiply(" + n1 + "," + n2 + ")");
       return n1 * n2;
}
```

RPC since 1983

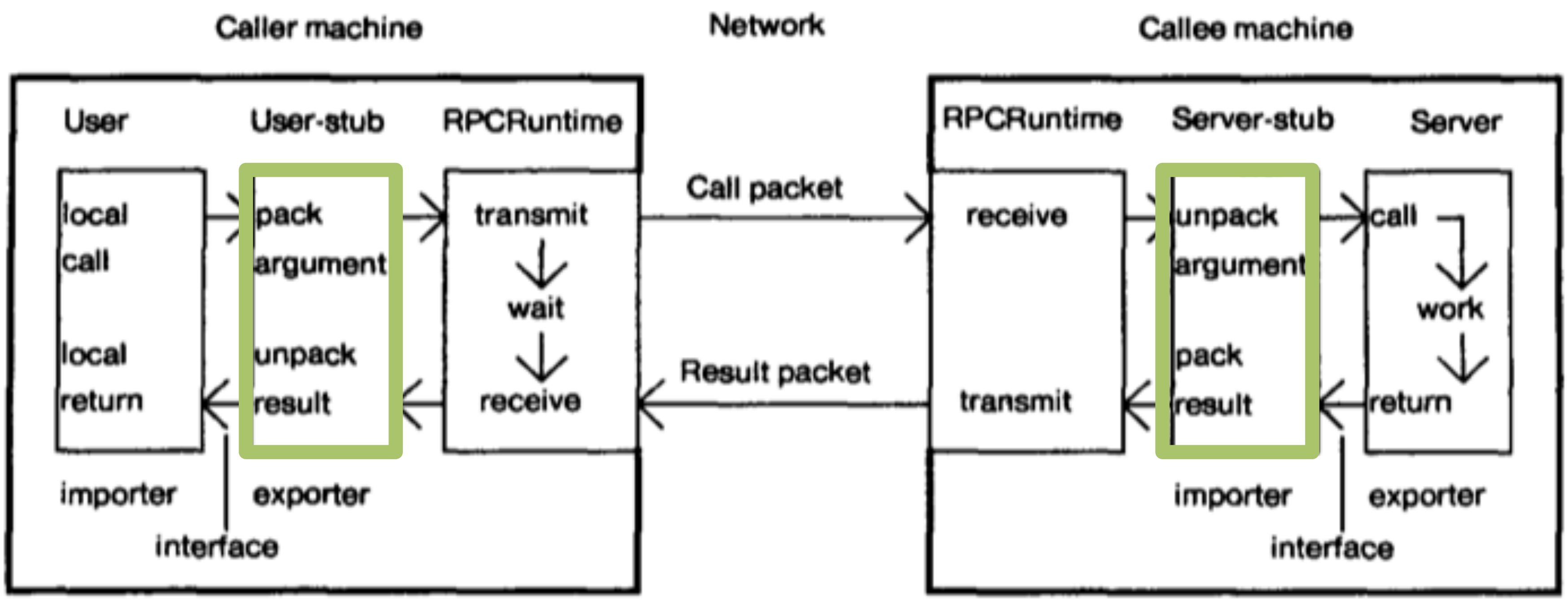
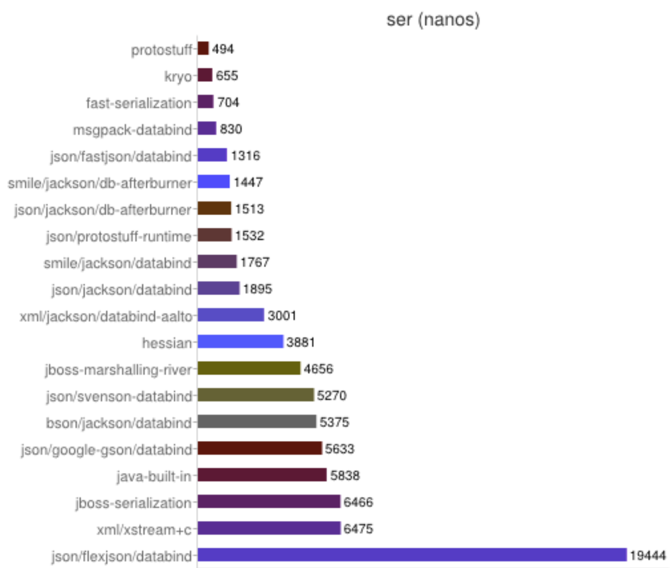


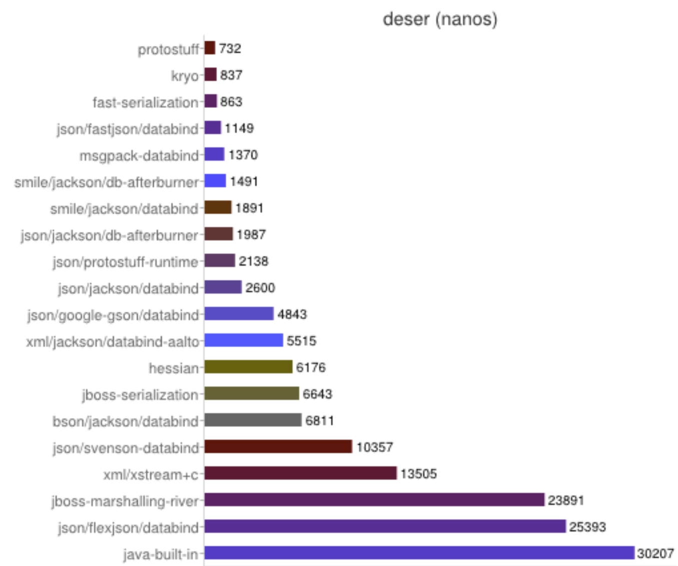
Fig. 1. The components of the system, and their interactions for a simple call.

Marshalling

Marshalling

- Format data into packets
  - Tricky for arrays, pointers, objects, ..
- Matters for performance
  - https://github.com/eishay/jvm-serializers/wiki





Other Details

- Binding
  - Client needs to find a server's networking address
  - Will cover in later classes
- Threading
  - Client need multiple threads, so have >1 call outstanding, match up replies to request
  - Handler may be slow, server also need multiple threads handling requests concurrently

RPC vs LPC

- 3 properties of distributed computing that make achieving transparency difficult:
  - Partial failures
  - Latency
  - Memory access

RPC Failures

- Request from cli → srv lost
- Reply from srv → cli lost
- Server crashes after receiving request
- Client crashes after sending request

Partial Failures

- In local computing:
  - if machine fails, application fails
- In distributed computing:
  - if a machine fails, part of application fails
  - one cannot tell the difference between a machine failure and network failure
- How to make partial failures transparent to client?

Strawman Solution

- Make remote behavior identical to local behavior:
  - Every partial failure results in complete failure
    - You abort and reboot the whole system
  - You wait patiently until system is repaired

- Problems with this solution:
  - Many catastrophic failures
  - Clients block for long periods
    - System might not be able to recover

RPC Exactly Once

- Impossible in practice
- Imagine that message triggers an external physical thing
  - E.g., a robot fires a nerf dart at the professor
- The robot could crash immediately before or after firing and lose its state. Don’t know which one happened. Can, however, make this window very small.

RPC At Least Once

- Ensuring at least once
  - Just keep retrying on client side until you get a response.
  - Server just processes requests as normal, doesn't remember anything. Simple!

- Is "at least once" easy for applications to cope with?
  - Only if operations are idempotent
  - x=5 okay
  - Bank -= $10 not okay

Possible semantics for RPC

- At most once
  - Zero, don't know, or once
- Server might get same request twice…
- Must re-send previous reply and not process request
  - Keep cache of handled requests/responses
  - Must be able to identify requests
  - Strawman: remember all RPC IDs handled.
    - Ugh! Requires infinite memory.
  - Real: Keep sliding window of valid RPC IDs, have client number them sequentially.

Implementation Concerns

- As a general library, performance is often a big concern for RPC systems
- Major source of overhead: copies and marshaling/unmarshaling overhead
- Zero-copy tricks:
  - Representation: Send on the wire in native format and indicate that format with a bit/byte beforehand. What does this do? Think about sending uint32 between two little-endian machines
  - Scatter-gather writes (writev() and friends)

Dealing with Environmental Differences

- If my function does: read(foo, ...)
- Can I make it look like it was really a local procedure call??
- Maybe!
  - Distributed filesystem...
- But what about address space?
  - This is called distributed shared memory
  - People have kind of given up on it - it turns out often better to admit that you're doing things remotely

Summary: Expose Remoteness to Client

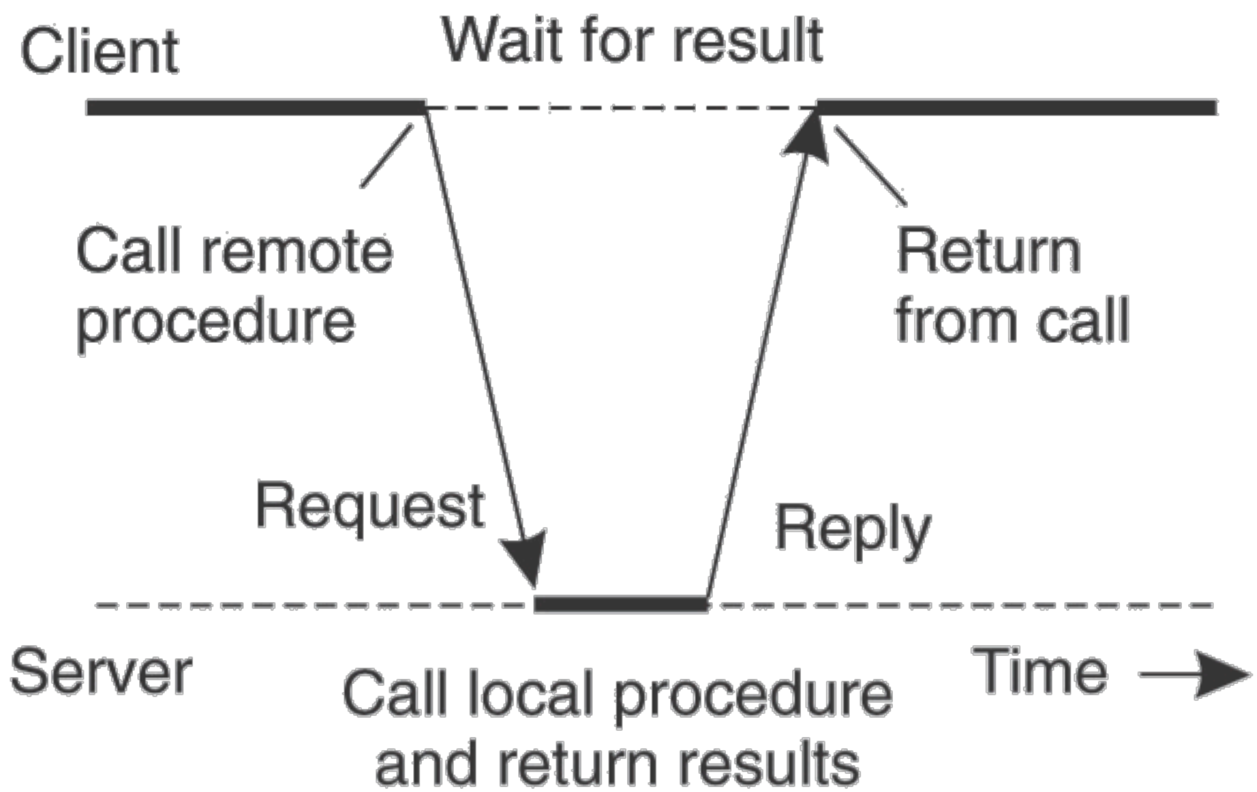
- Expose RPC properties to client, since you cannot hide them
- Application writers have to decide how to deal with partial failures
  - Consider: E-commerce application vs. game

Important Lessons

- Procedure calls
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Not only way…
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
- How to deal with hard problem
  - Give up and let programmer deal with it

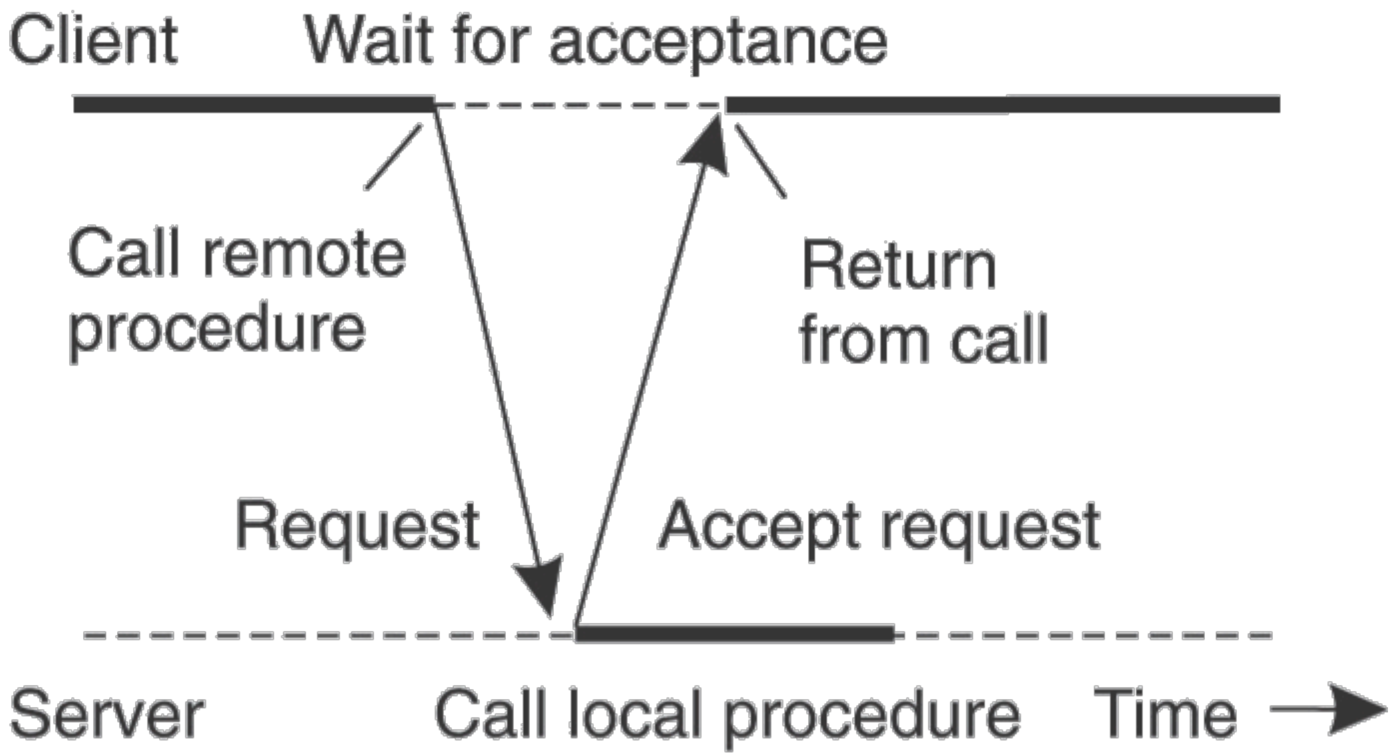
Bonus Topic 1: Sync vs. Async

Synchronous RPC



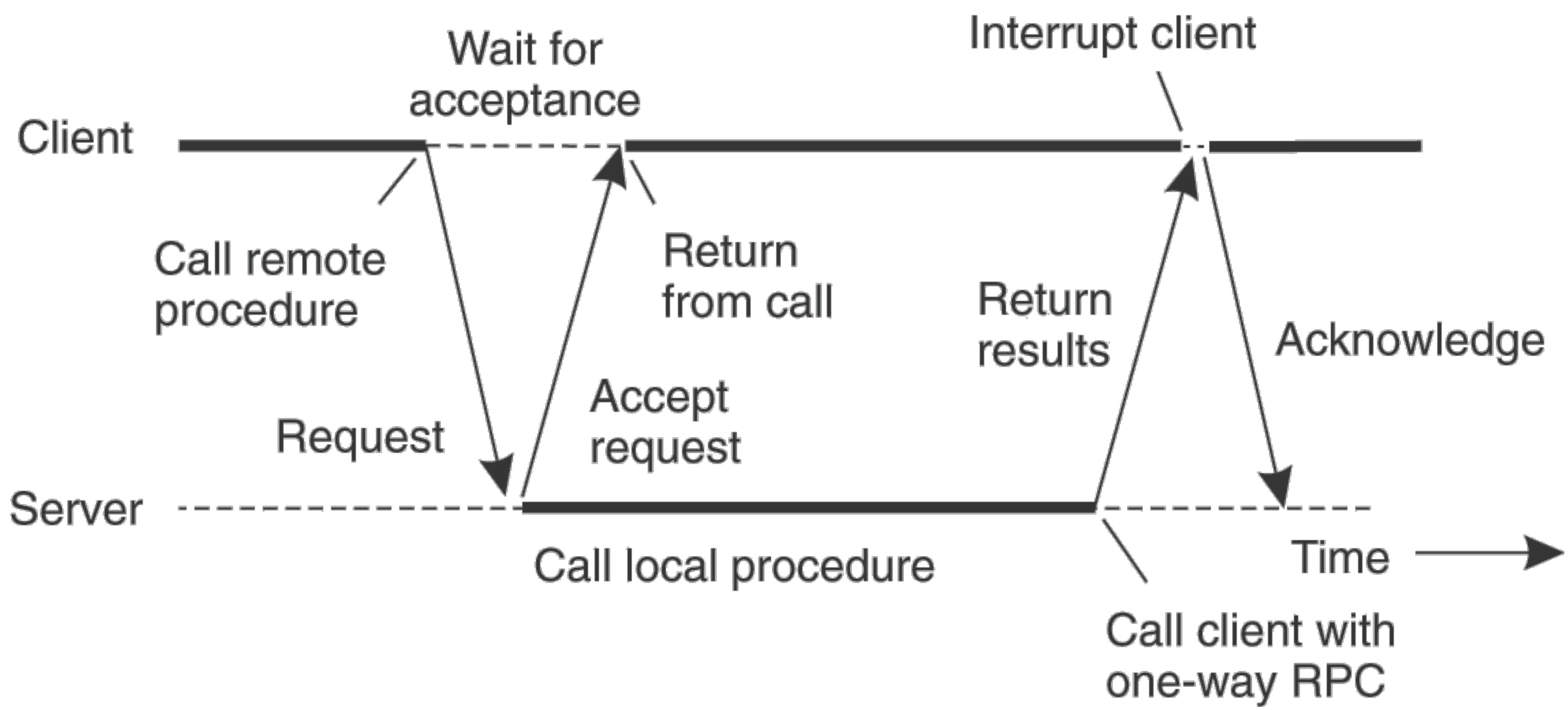
The interaction between client and server
in a traditional RPC.

Asynchronous RPC



The interaction using asynchronous RPC

Asynchronous RPC



A client and server interacting through two asynchronous RPCs.

Bonus Topic 2:
How Fast?

Implementing RPC Numbers

Table I. Performance Results for Some Examples of Remote Calls

Procedure	Minimum	Median	Transmission	Local-only
no args/results	1059	1097	131	9
1 arg/result	1070	1105	142	10
2 args/results	1077	1127	152	11
4 args/results	1115	1171	174	12
10 args/results	1222	1278	239	17
1 word array	1069	1111	131	10
4 word array	1106	1153	174	13
10 word array	1214	1250	239	16
40 word array	1643	1695	566	51
100 word array	2915	2926	1219	98
resume except'n	2555	2637	284	134
unwind except'n	3374	3467	284	196

Results in microseconds

COPS RPC Numbers

System	Operation	Latency (ms) 50%	99%	99.9%
Thrift	ping	0.26	3.62	12.25
COPS	get_by_version	0.37	3.08	11.29
COPS-GT	get_by_version	0.38	3.14	9.52
COPS	put_after (1)	0.57	6.91	11.37
COPS-GT	put_after (1)	0.91	5.37	7.37
COPS-GT	put_after (130)	1.03	7.45	11.54

Bonus Topic 3:
Modern Feature Sets

Modern RPC features

- RPC stack generation (some)
- Many language bindings
- No service binding interface
- Encryption (some?)
- Compression (some?)

Intermission

MapReduce

- Distributed Computation

Why Distributed Computations?

- How long to sort 1 TB on one computer?
  - One computer can read ~30MBps from disk
    - 33 000 secs => 10 hours just to read the data!

- Google indexes 100 billion+ web pages
  - 100 * 10^9 pages * 20KB/page = 2 PB

- Large Hadron Collider is expected to produce 15 PB every year!

Solution: Use Many Nodes!

- Data Centers at Amazon/Facebook/Google
  - Hundreds of thousands of PCs connected by high speed LANs
- Cloud computing
  - Any programmer can rent nodes in Data Centers for cheap
- The promise:
  - 1000 nodes ➔ 1000X speedup

Distributed Computations are Difficult to Program

- Sending data to/from nodes
- Coordinating among nodes
- Recovering from node failure
- Optimizing for locality
- Debugging

Same for all problems

MapReduce

- A programming model for large-scale computations
  - Process large amounts of input, produce output
  - No side-effects or persistent state

- MapReduce is implemented as a runtime library:
  - automatic parallelization
  - load balancing
  - locality optimization
  - handling of machine failures

MapReduce design

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files

More Specifically…

- Programmer specifies two methods:
  - map(k, v) → <k', v'>*
  - reduce(k', <v'>*) → <k', v'>*
- All v' with same k' are reduced together

- Usually also specify:
  - partition(k’, total partitions) -> partition for k’
    - often a simple hash of the key
    - allows reduce operations for different k’ to be parallelized

Example: Count word frequencies in web pages

- Input is files with one doc per record
- Map parses documents into words
  - key = document URL
  - value = document contents
- Output of map:

"doc1", "to be or not to be" →

"to", "1"
"be", "1"
"or", "1"
...

Example: word frequencies

- Reduce: computes sum for a key

key = “be”
values = “1”, “1”

“2”

key = “not”
values = “1”

“1”

key = “or”
values = “1”

“1”

key = “to”
values = “1”, “1”

“2”

- Output of reduce saved

“be”, “2”
“not”, “1”
“or”, “1”
“to”, “2”

Example: Pseudo-code

```
Map(String input_key, String input_value):
  //input_key: document name
  //input_value: document contents
  for each word w in input_values:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator intermediate_values):
  //key: a word, same for input and output
  //intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce is widely applicable

- Distributed grep
- Document clustering
- Web link graph reversal
- Detecting duplicate web pages
- …

MapReduce implementation

- Input data is partitioned into M splits
- Map: extract information on each split
  - Each Map produces R partitions
- Shuffle and sort
  - Bring M partitions to the same reducer
- Reduce: aggregate, summarize, filter or transform
- Output is in R result files, stored in a replicated, distributed file system (GFS).

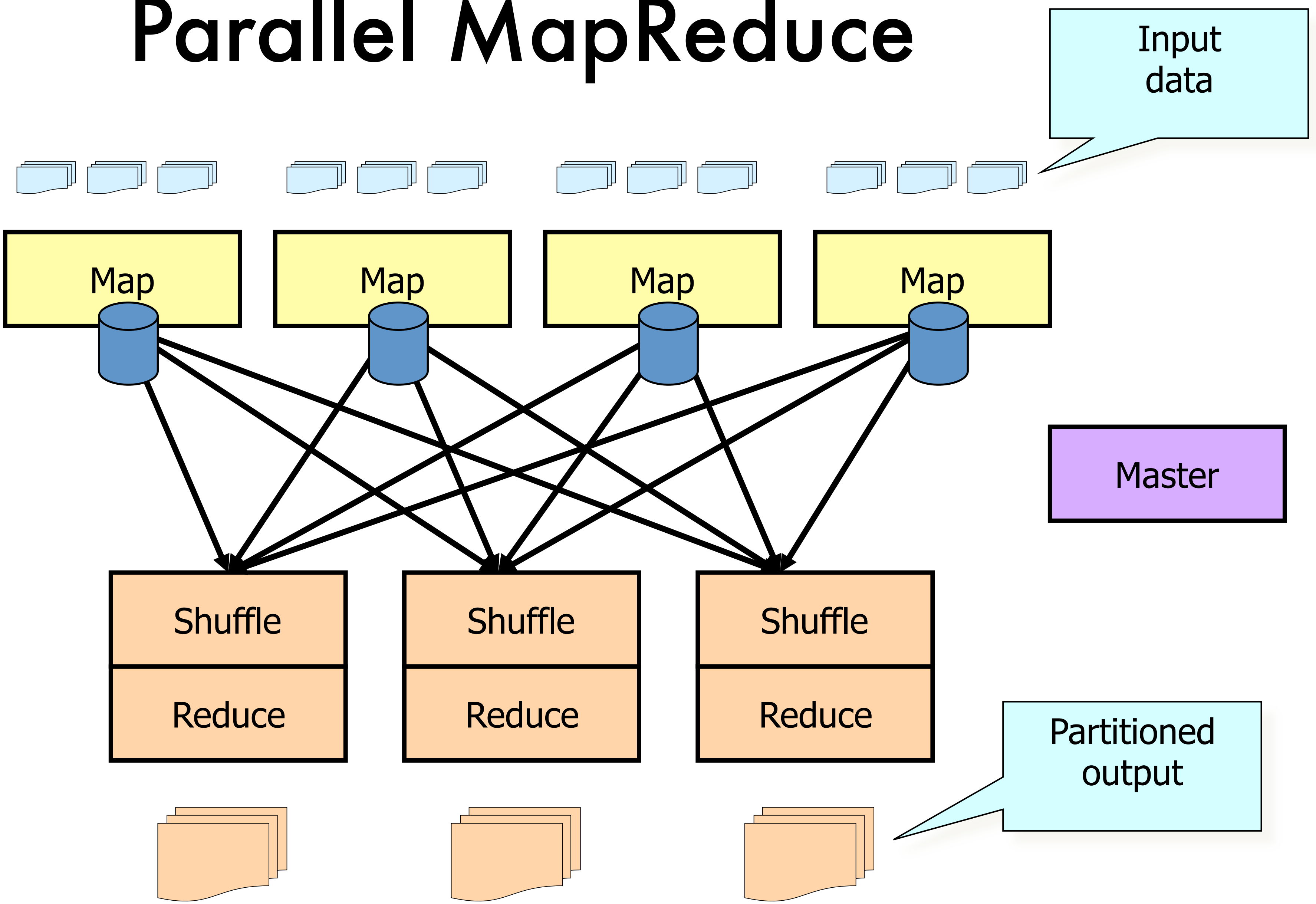
MapReduce scheduling

- One master, many workers
  - Input data split into *M* map tasks
  - *R* reduce tasks
  - Tasks are assigned to workers dynamically
- Assume 1000 workers, what's a good choice for M & R?
  - M > #workers, R > #workers
  - Master's scheduling efforts increase with M & R
    - Practical implementation : O(M*R)
  - E.g. *M*=100,000; *R*=2,000; workers=1,000

MapReduce scheduling

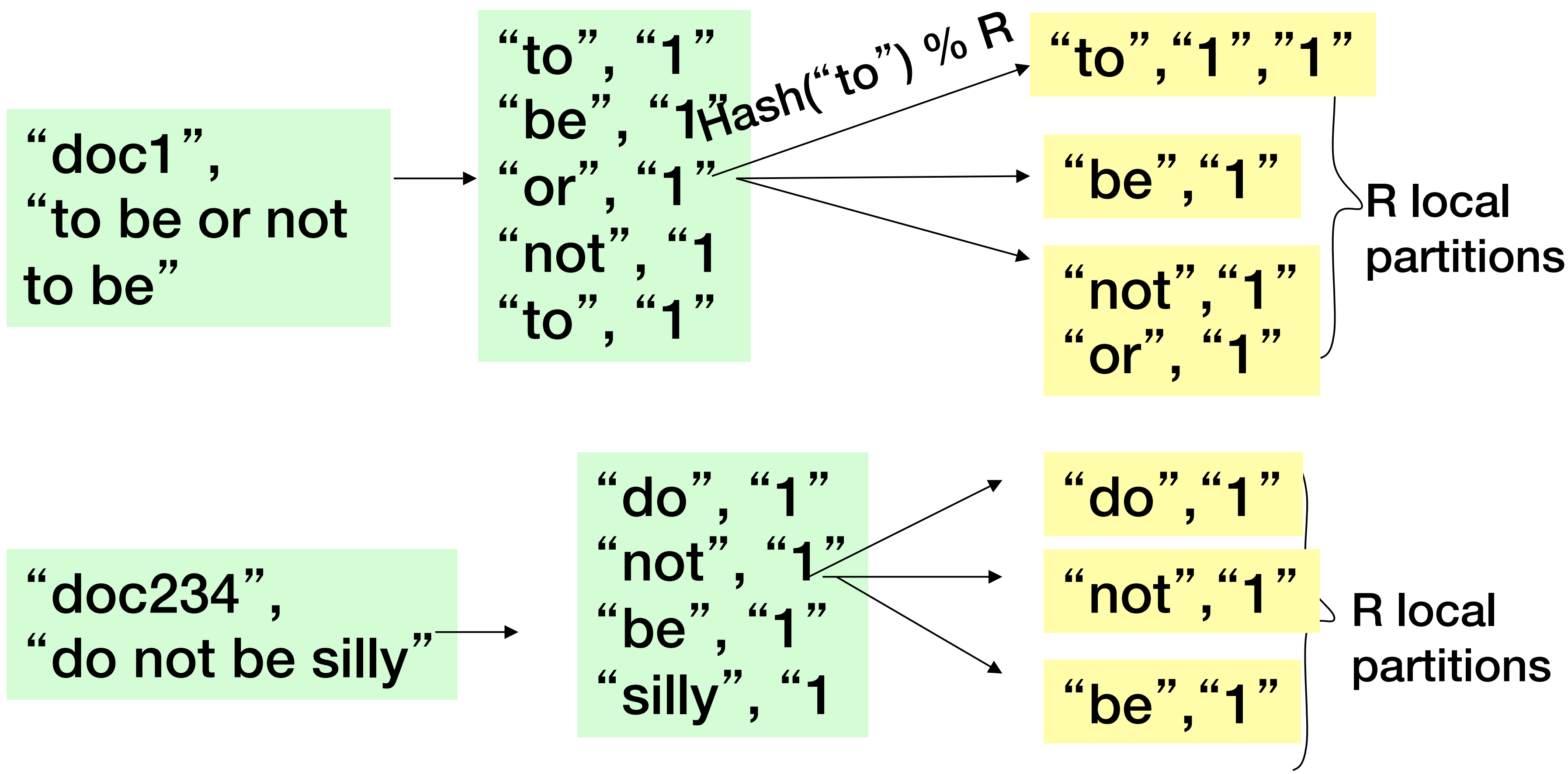
- Master assigns a map task to a free worker
  - Prefers “close-by” workers when assigning task
  - Worker reads task input (often from local disk!)
  - Worker produces R **local files** containing intermediate k/v pairs
- Master assigns a reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user’s *Reduce* op to produce the output

Parallel MapReduce



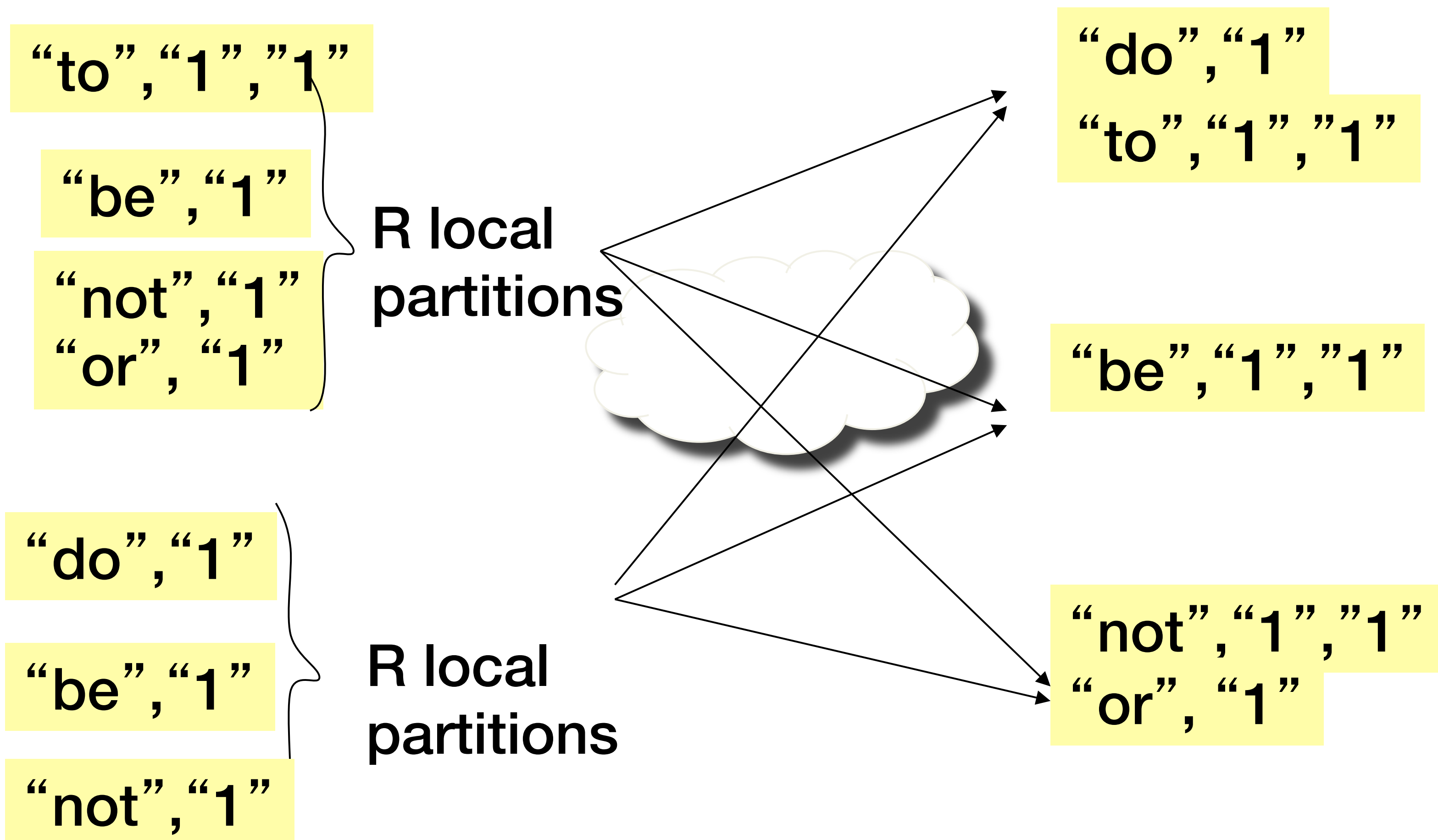
WordCount Internals

- Input data is split into M map jobs
- Each map job generates in R local partitions



WordCount Internals

- Shuffle brings same partitions to same reducer



WordCount Internals

- Reduce aggregates sorted key values pairs

“do”,“1”
“to”,“1”,”1” → “do”,“1”
“to”, “2”

“be”,“1”,”1” → “be”,“2”

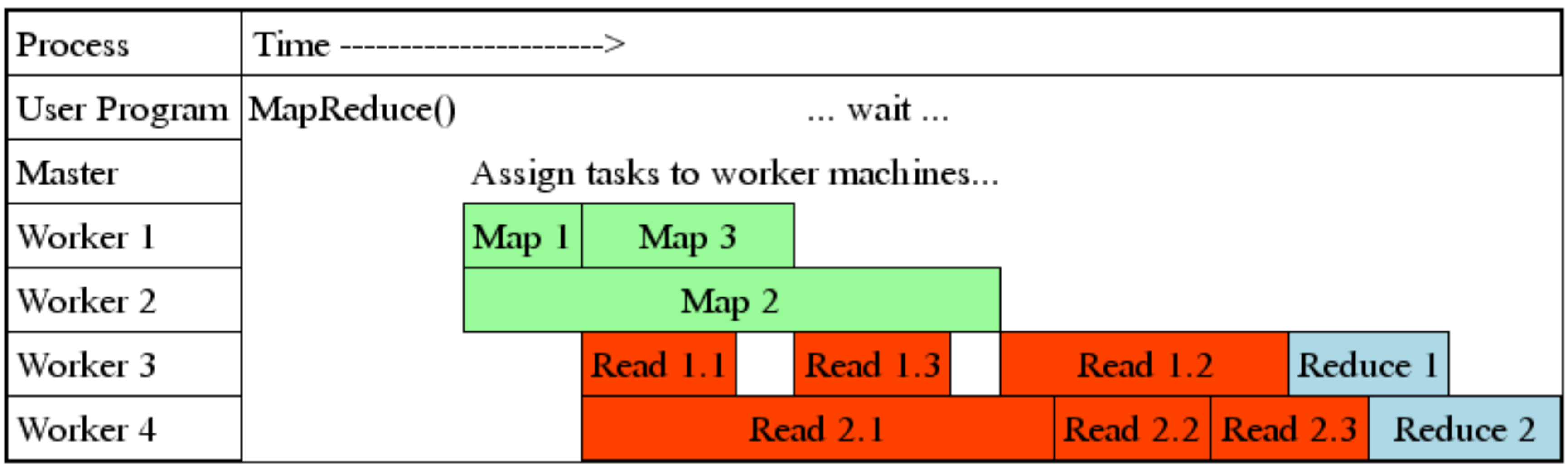
“not”,“1”,”1”
“or”, “1” → “not”,“2”
“or”, “1”

The importance of partition function

- partition(k', total partitions) -> partition for k'
  - e.g. hash(k') % R
- What is the partition function for sort?

Load Balance and Pipelining

- Fine granularity tasks: many more map tasks than machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution



Fault tolerance via re-execution

On worker failure:

- Re-execute completed and in-progress map tasks
- Re-execute in-progress reduce tasks
- Task completion committed through master

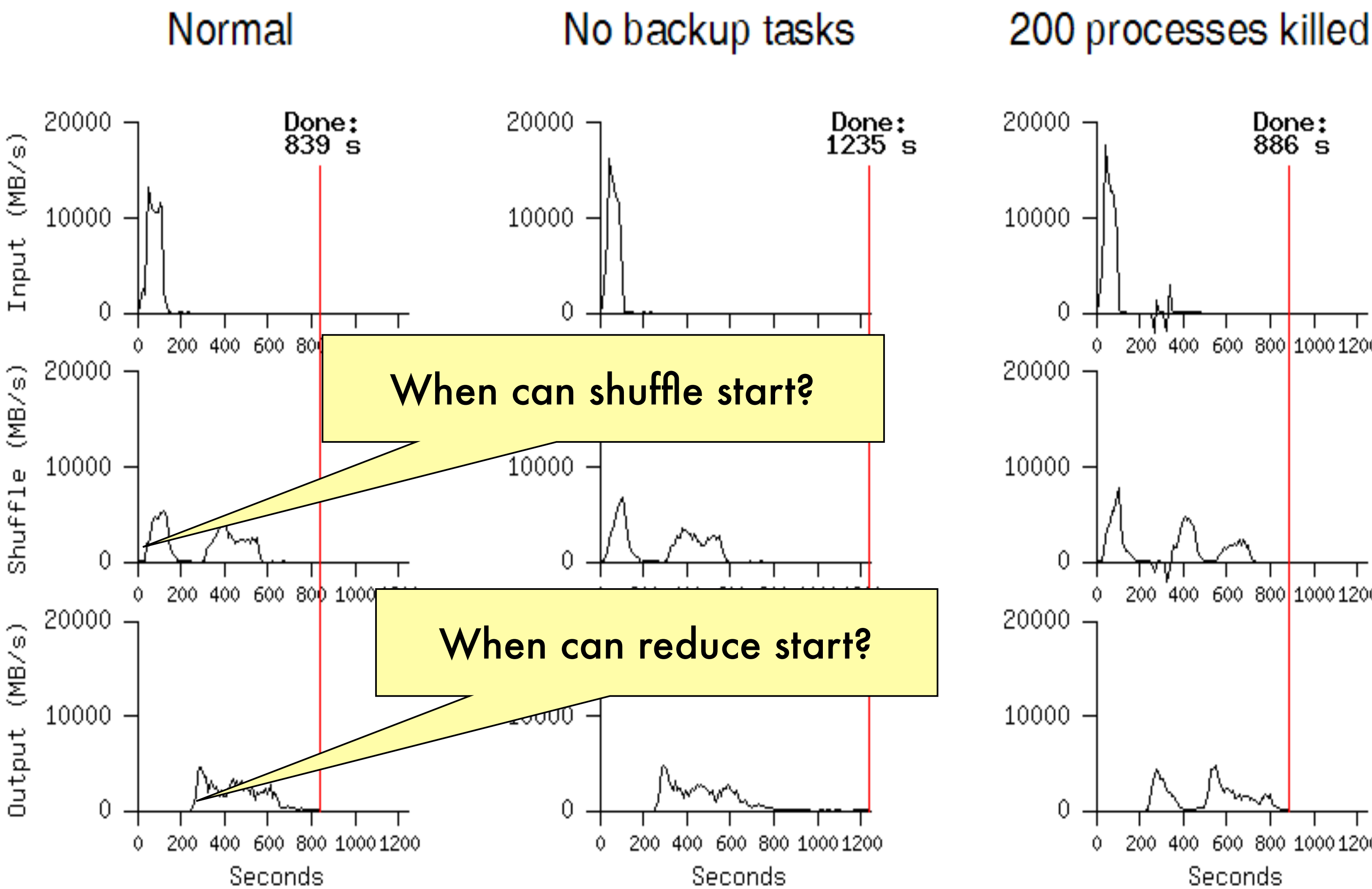
On master failure:

- State is checkpointed to GFS: new master recovers & continues

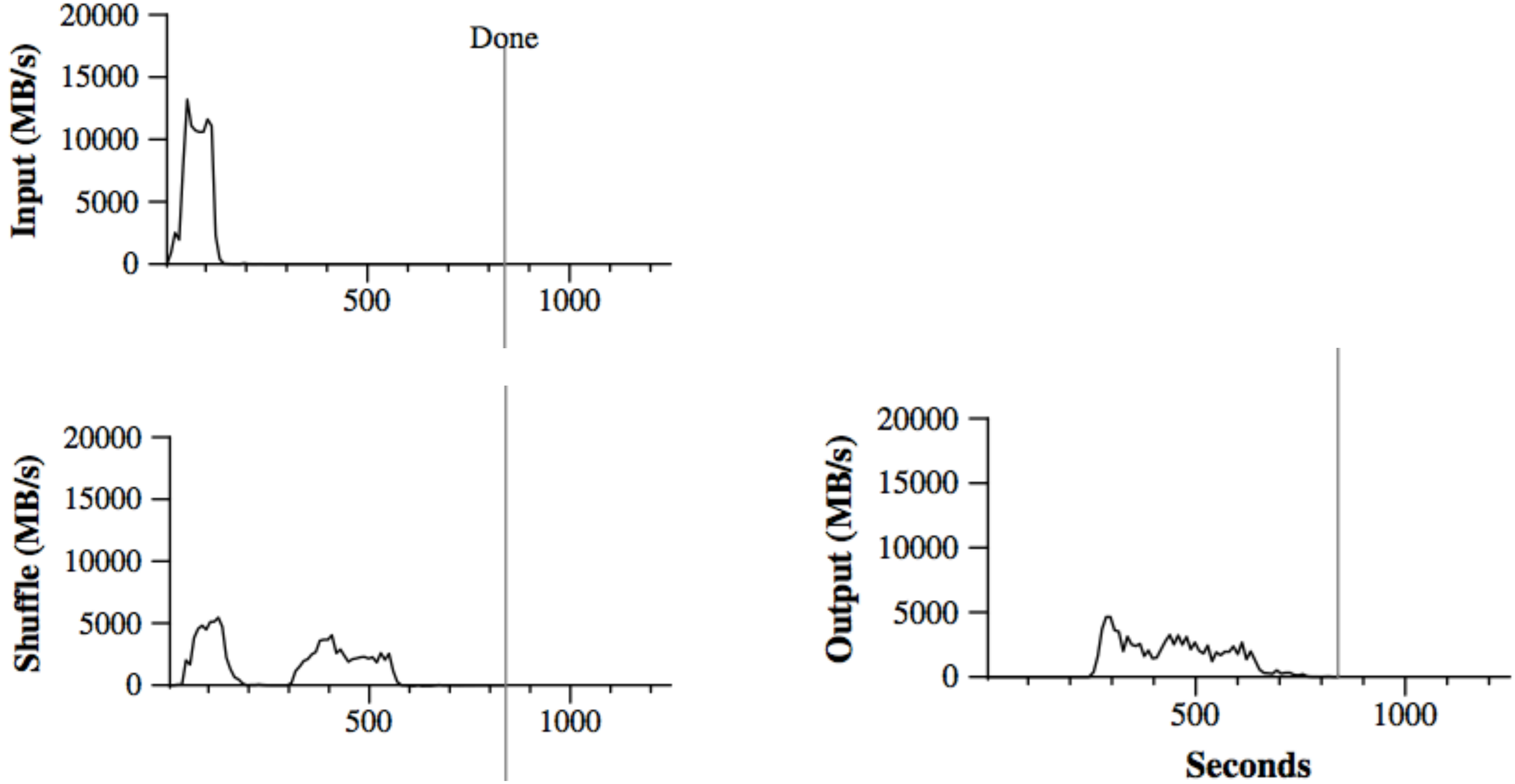
MapReduce Sort Performance

- 1TB (100-byte record) data to be sorted
- ~1800 machines
- M=15000 R=4000

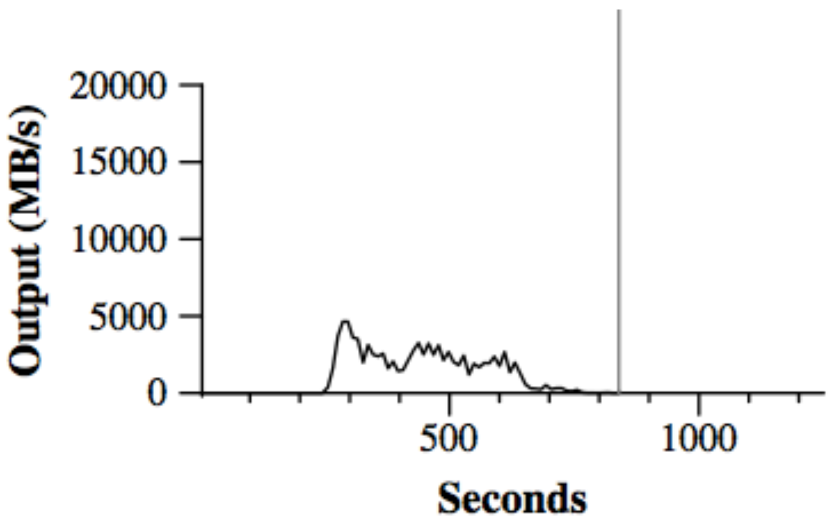
MapReduce Sort Performance



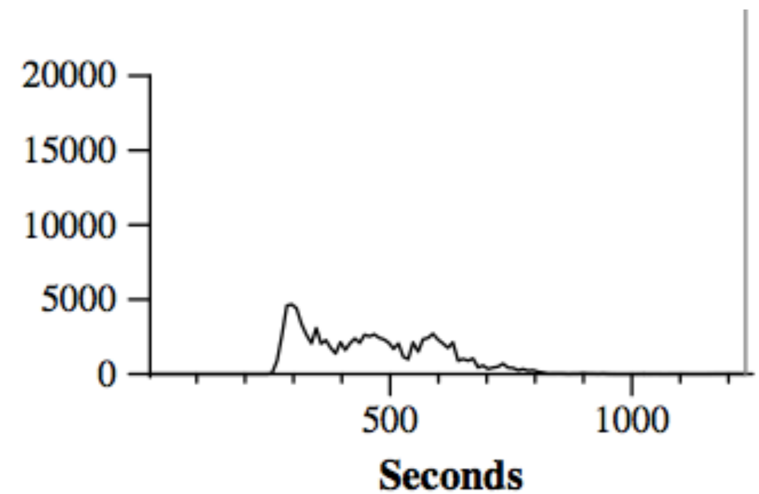
MapReduce Sort Performance (Normal Execution)



Effect of Backup Tasks



(a) Normal execution



(b) No backup tasks

Avoid straggler using backup tasks

- Slow workers drastically increase completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
  - An unusually large reduce partition
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

Refinements

- Combiner
  - Partial merge of the results before transmission
  - “Map-side reduce”
    - Often code for combiner and reducer is the same
- Skipping Bad Records
  - Signal handler catches seg fault/bus error
  - Send “last gasp” udp packet to master
  - If the master gets N “last gasp” for the same record it marks it to be skipped on future restarts