Review

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/rong_chen

Syllabus

Basic Techniques

- RPC/JNI
- Sequential/Release
- Eventual/Causal

Fault Tolerance

- Recovery: logging
- Concurrency: 2PL/SI
- Commit: 2PC
- Consensus: Paxos/RSM
- Byzantine Fault Tolerance

Distributed Computation

- Data-parallel: MR/Dryad
- Advance: Graph/Matrix

Distributed Storage

- DFS: GFS/Chubby
- NoSQL: BigTable/Dynamo

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

– Leslie Lamport

Remote Procedure Calls

Problems with the Sockets API

The sockets interface forces a read/write mechanism

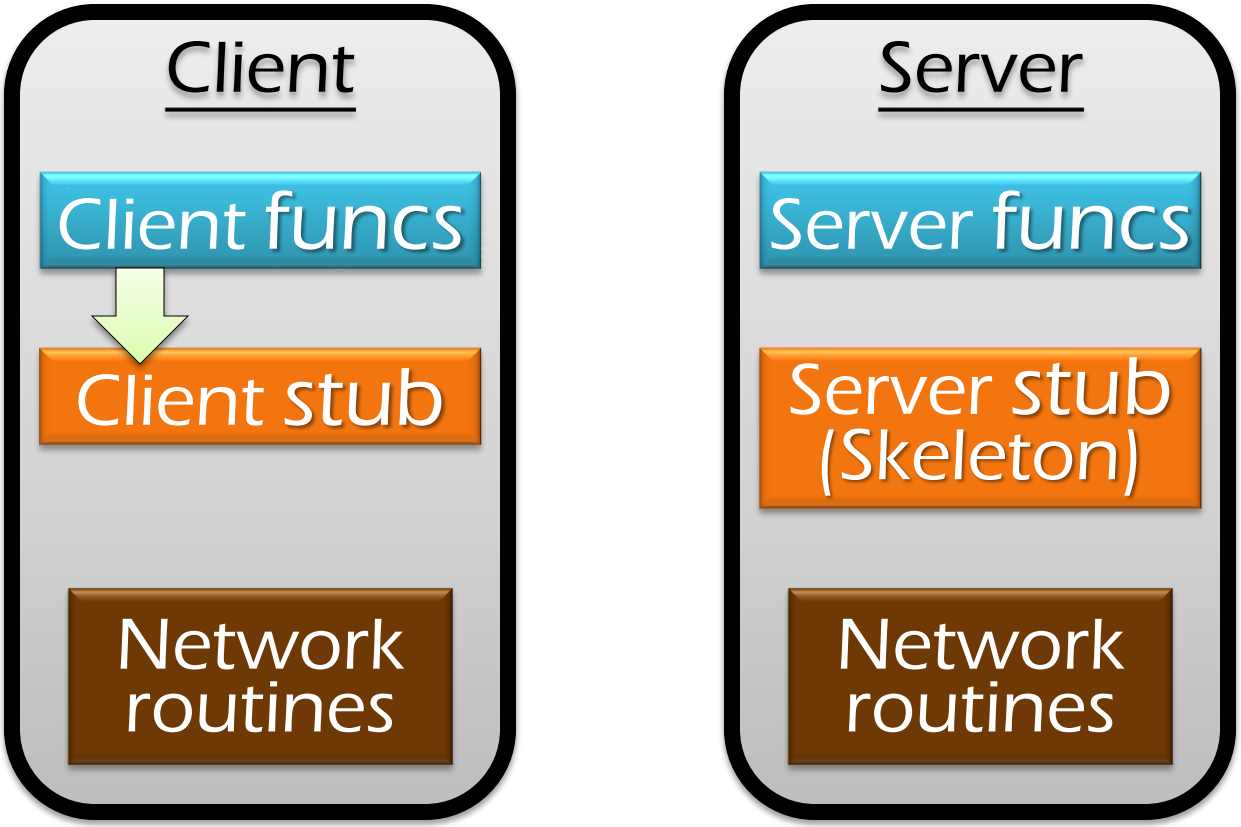
Programming is often easier with a functional interface

To make distributed computing look more like centralized computing

I/O (read/write) is not the way to go

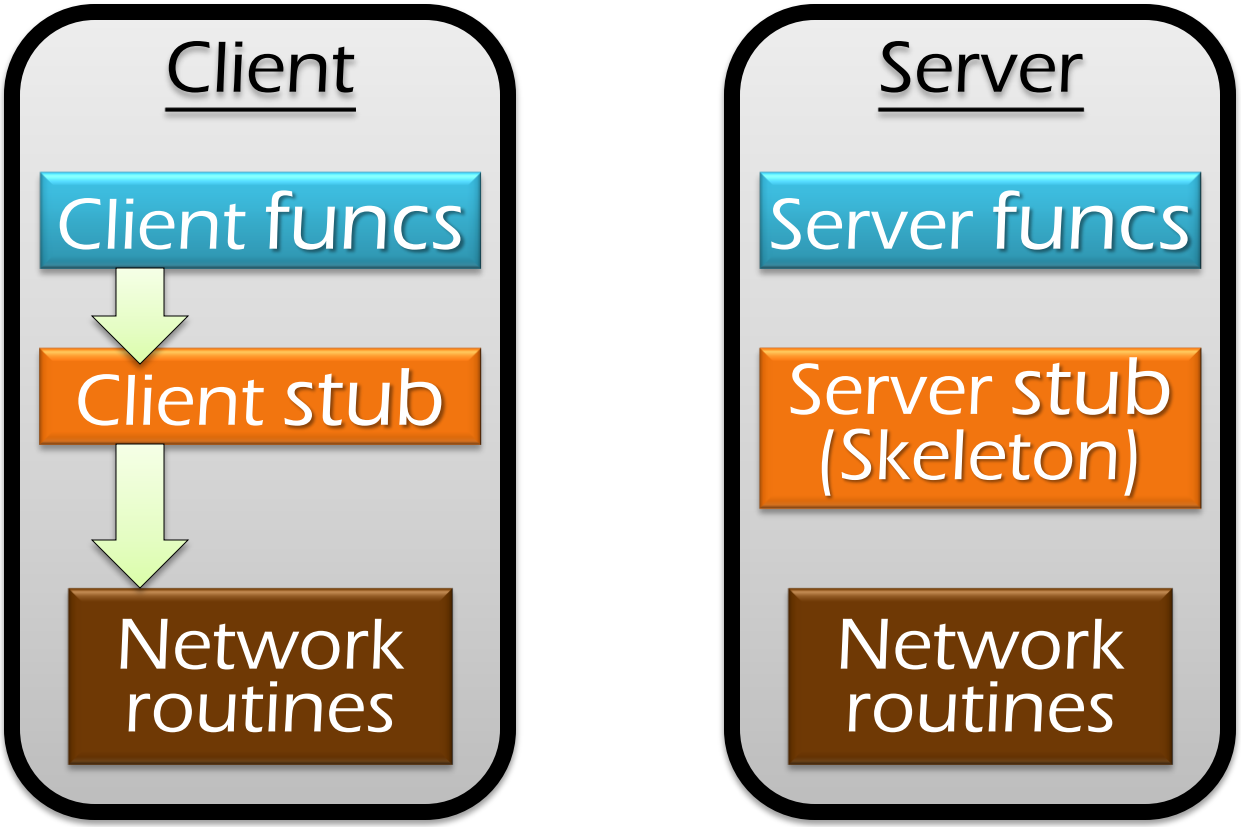
Stub Functions

1. Client calls **stub** (params on stack)



Stub Functions

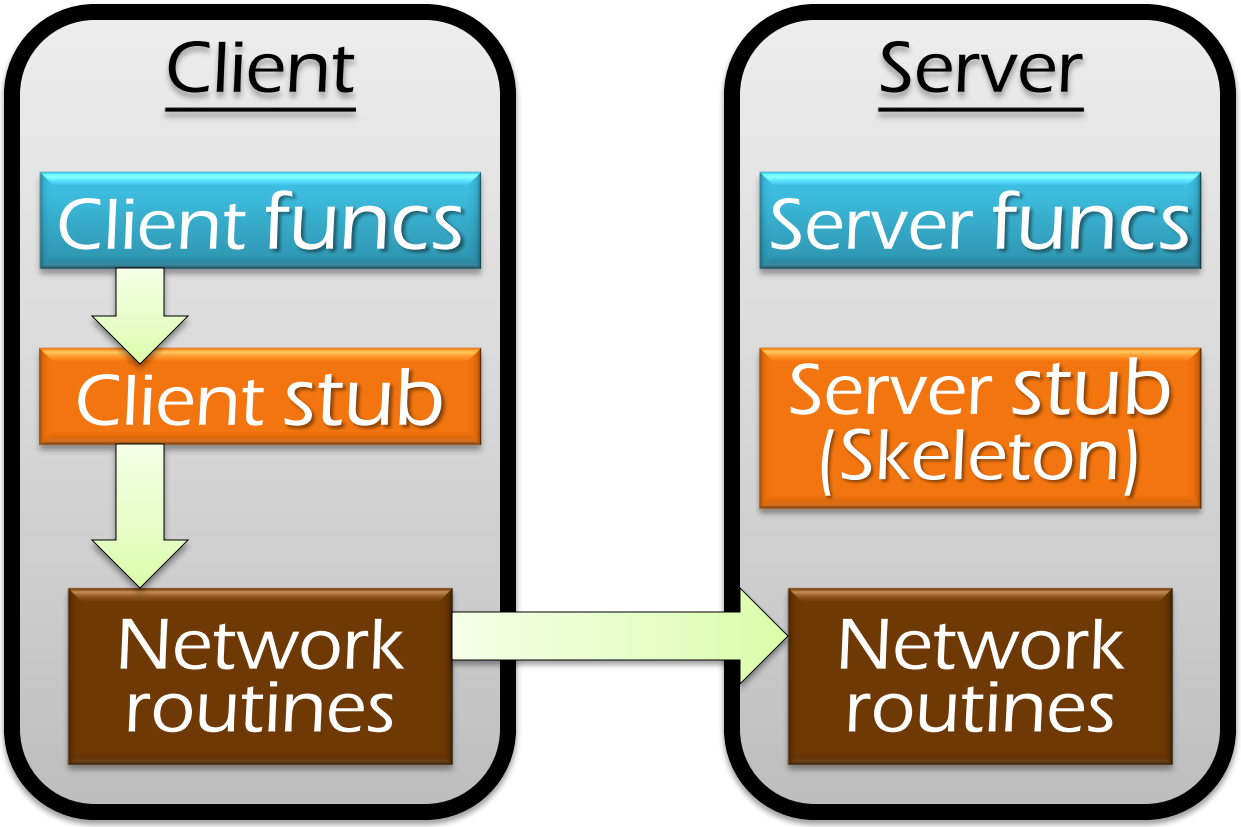
2. **Stub** marshals params to net **message**



Marshals = put data in a form suitable for transmission over a network

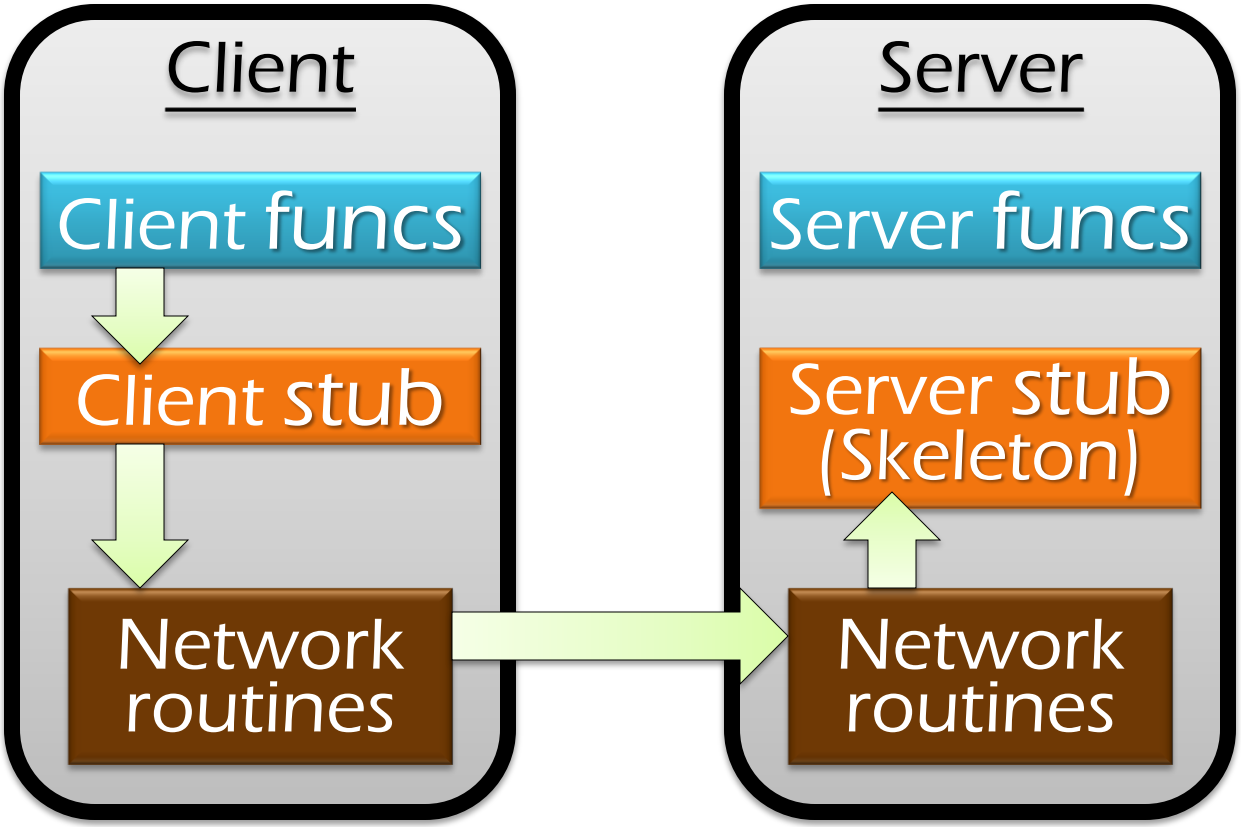
Stub Functions

3. Network routines sends **message** to server



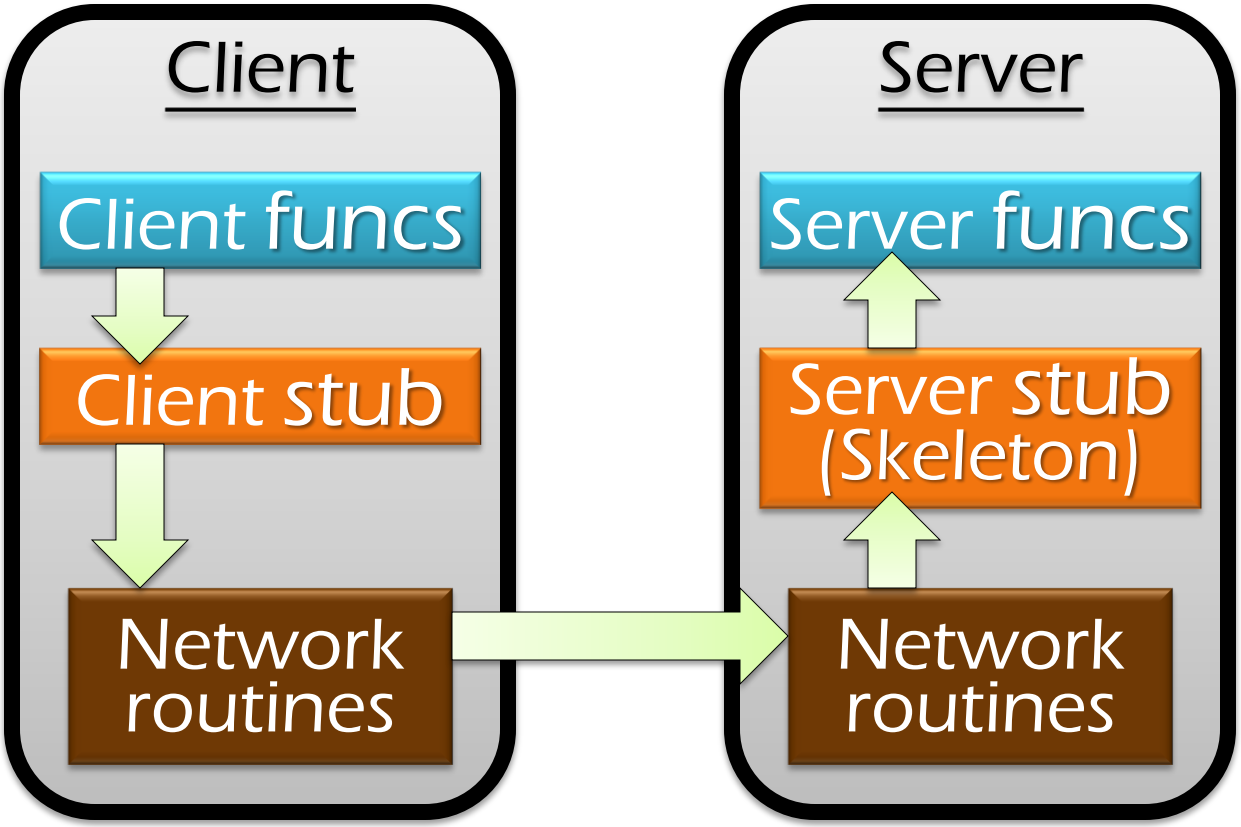
Stub Functions

4. Receive message: send it to server stub



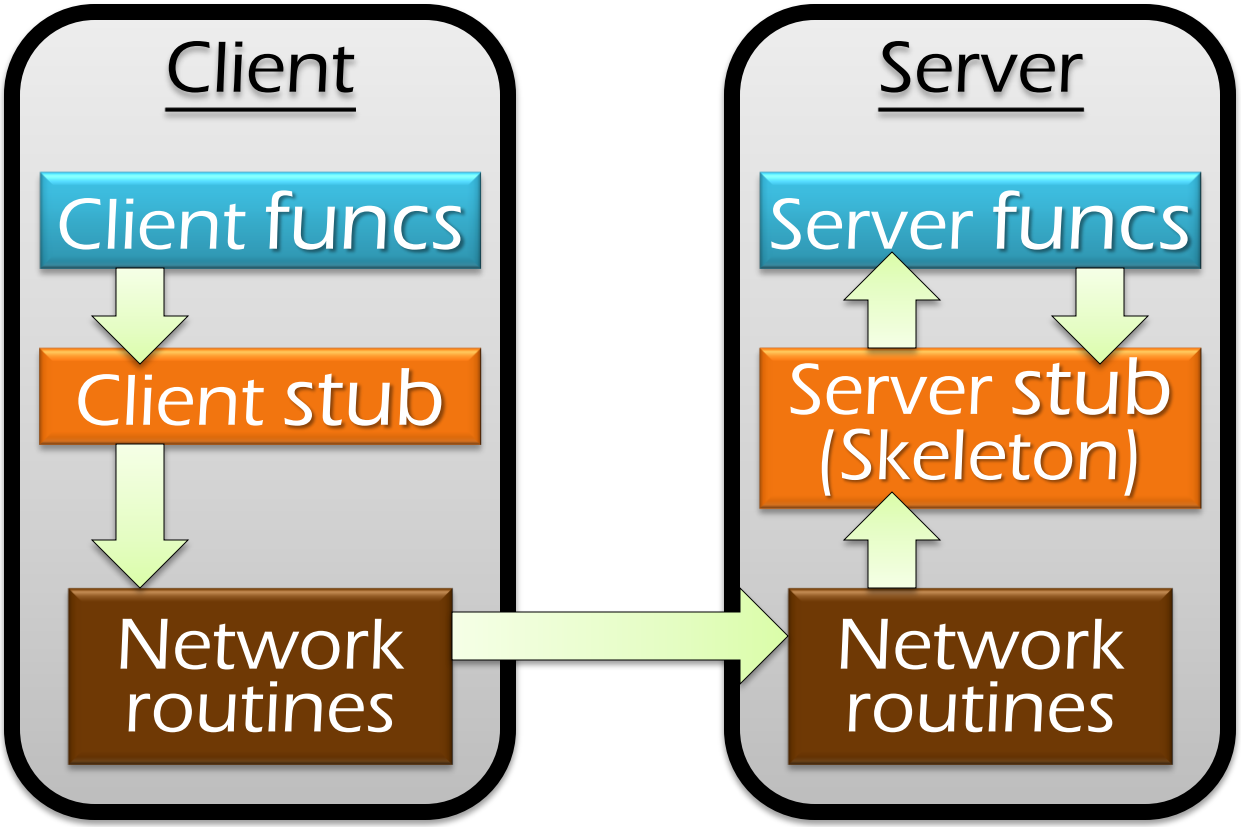
Stub Functions

5. Unmarshal parameters, call server function



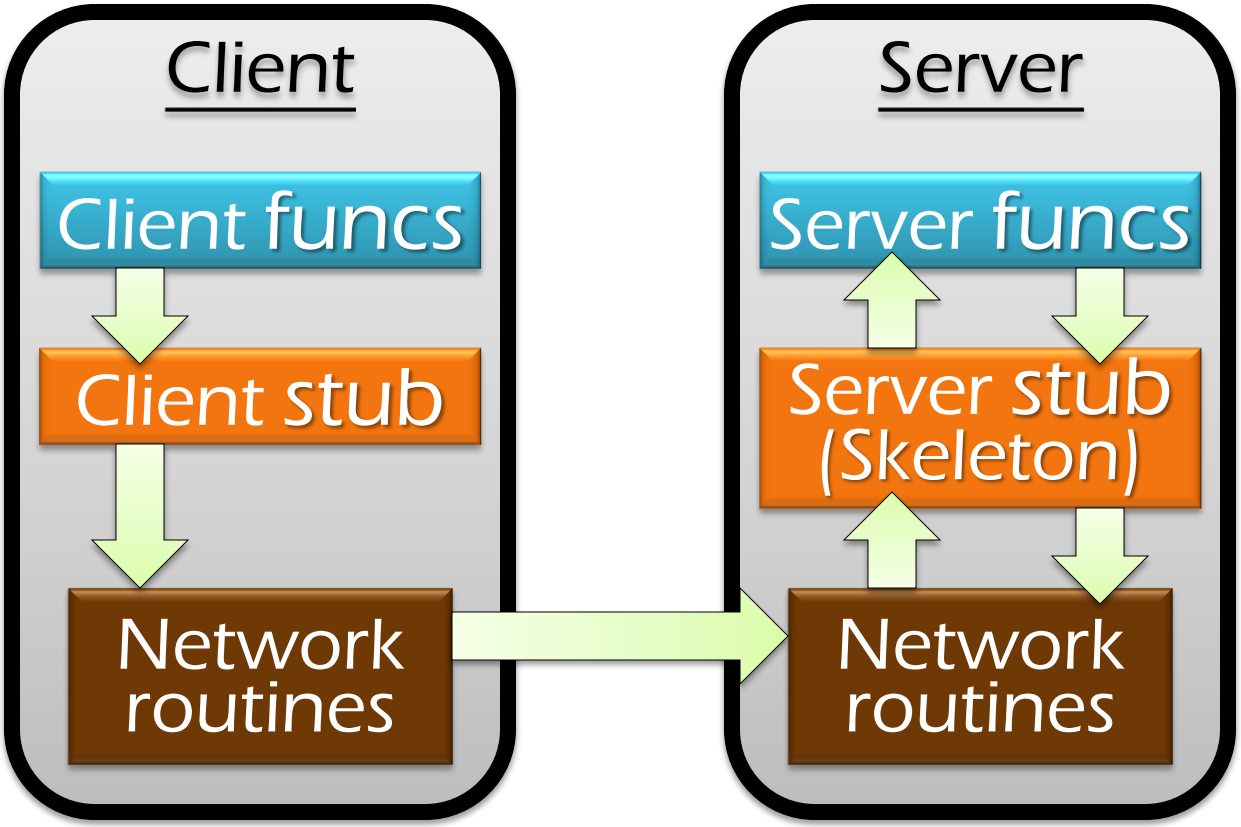
Stub Functions

6. Return from server function



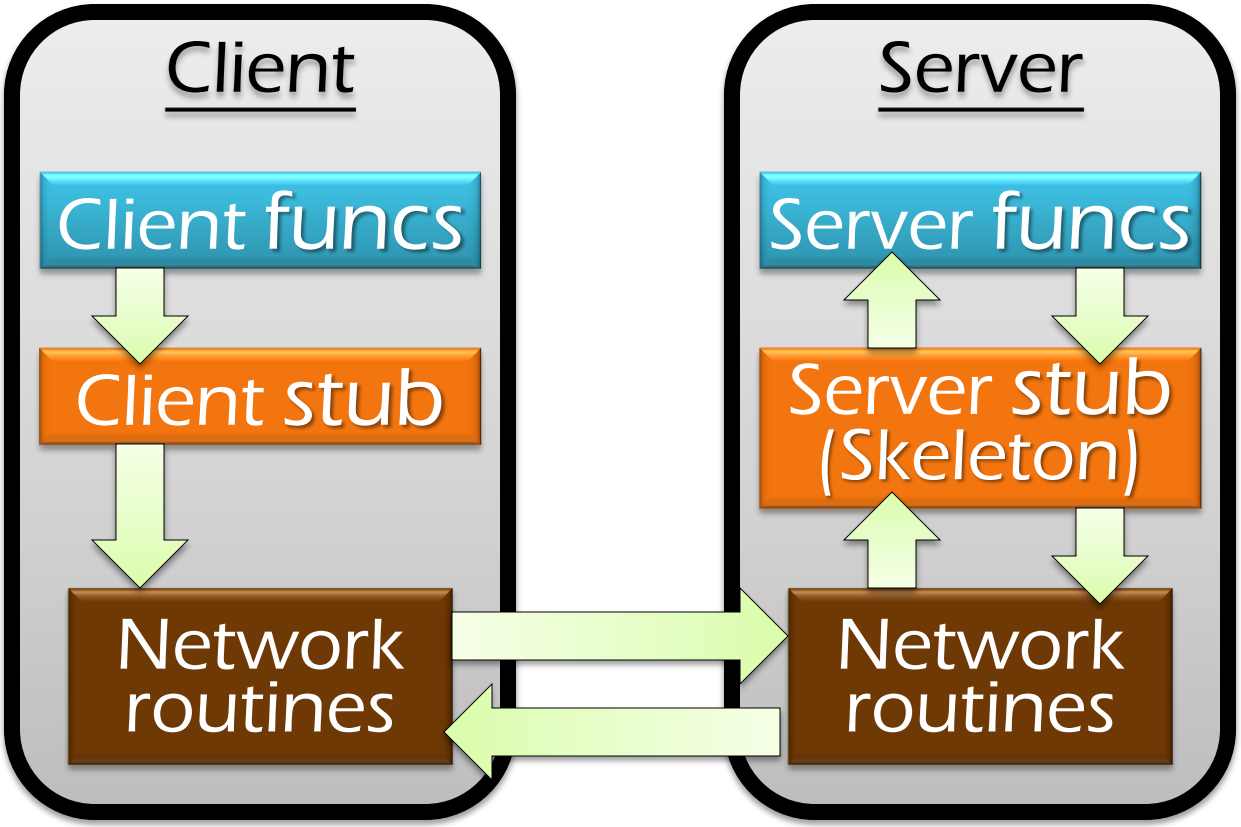
Stub Functions

7. Marshal return value and send message



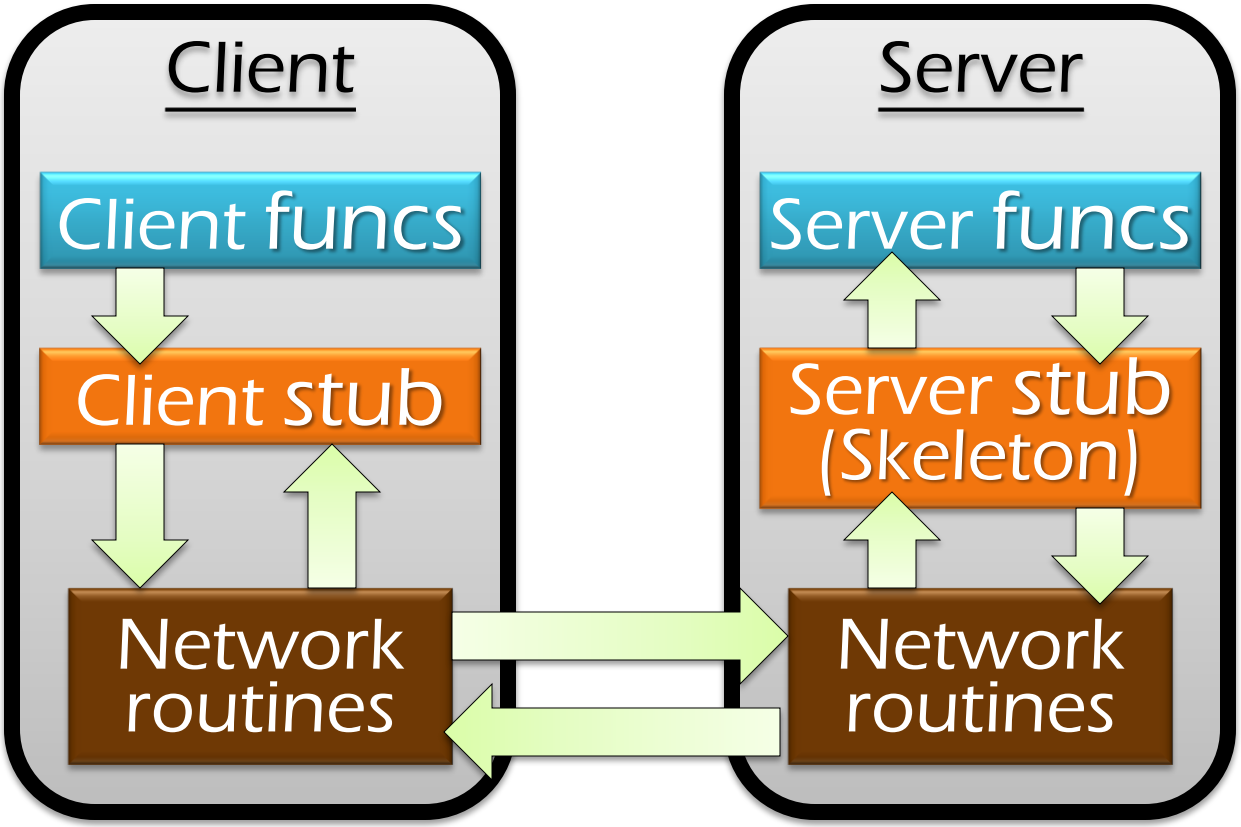
Stub Functions

8. Transfer **message** over network



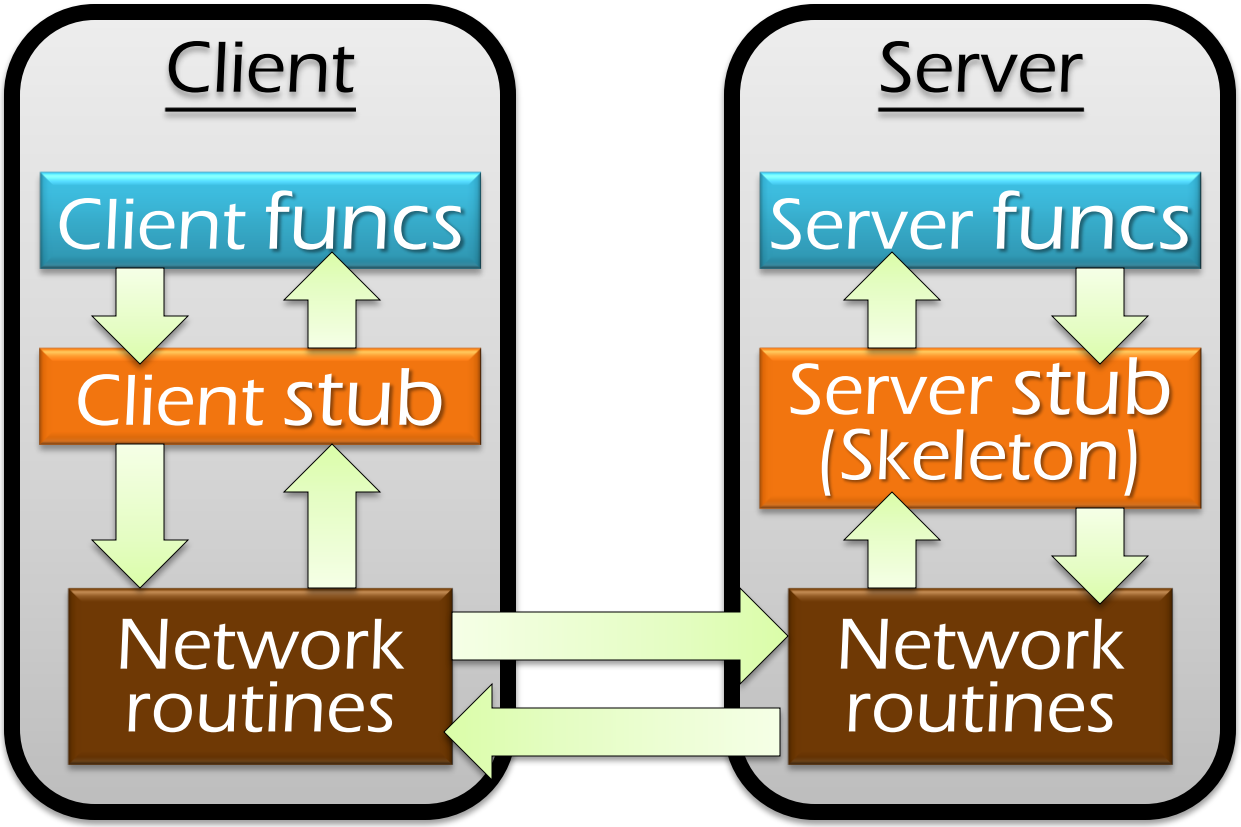
Stub Functions

9. Receive message: client stub is receiver



Stub Functions

10. Unmarshal return value, return to client code



Benefits

Procedure call interface

Writing application is simplified

- RPC hides all network code into stub function
- Application programmers don't have to worry about details
  - Sockets, port numbers, byte ordering

Consistency

What is Consistency?

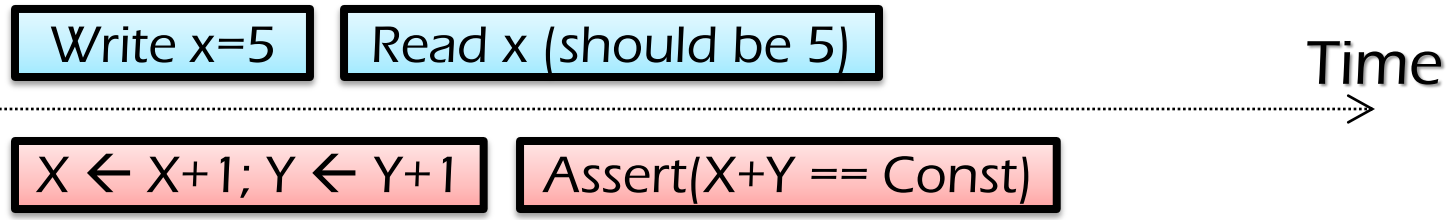
Consistency model defines **rules** for the apparent order and visibility of **updates**, and it is a continuum with **tradeoffs**

– Todd Lipcon

Examples

Single object consistency is also called "**coherence**"

Local memory:



Database:

Consistency across multiple objects (all or nothing)

Consistency Challenges

No right or wrong consistency models

- Tradeoff between ease of programmability and efficiency
  (*e.g.*, what's the consistency model for web pages?)

Consistency is hard in (distributed) systems

- Data replication (Caching)
- Concurrency (no shared clock)
- Failures (machine or network)

Sequential Consistency

Strict consistency is not practical

- No global wall-clock available

Sequential consistency is the closest

Consistency rules

"global wall-clock" → "total order" of ops

1. Each CPUs' ops appear in order
2. All CPUs see results according to total order (*i.e.*, reads see most recent writes)

Sequential consistency is also intuitive results (just use the **total order** as **timestamp**)

Release Consistency

Introduce a special type of variables, called synchronization variables or locks

Locks can be acquired/released, not read/written

- Denoted by **acquire(L)** and **release**(L)

No more than one process can hold a lock L

- Hold a lock
  → acquired a lock and has not released it

K. Gharachorloo, et al. *Memory consistency and event ordering in scalable shared-memory multiprocessors*. ISCA, 1990

Sequential vs. Eventual

Sequential: pessimistic conflict handling

- Conflicting writes is **common** case
- Updates cannot take effect unless they are serialized **first**

Eventual: optimistic conflict handling

- Conflicting writes is **rare** case
- Let updates happen, worry about whether they can be serialized **later**

Eventual Consistency

For better **performance**

- Accept a write before being able to serialize it
- Reads can return a (possible) **stale** value than blocking for the **latest** value

Anomalies

- **Write-write** conflict
- **Stale** read (Loss of **causality**)

Eventual **convergence** of state

Causality preserving

Fault Tolerance

ACID Transactions

A (Atomicity)	**All-or-nothing** with respect to failures
C (Consistency)	Transactions maintain any internal state **invariants**
I (Isolation)	Concurrently executing transactions don't **interfere**
D (Durability)	Effect to transactions **survive** failures

What is ACID ?

T1 Transfer $1000 from A to B

T2 Transfer $500 from B to A

A T1 **completes** or **nothing** (ditto for T2)

C Preserves **invariants**, e.g. account balance > 0

I No **race**, as if T1 happens either before or after T2

D Once T1/T2 commits, stays done, no updates **lost**

All-or-nothing Atomicity

All-or-nothing

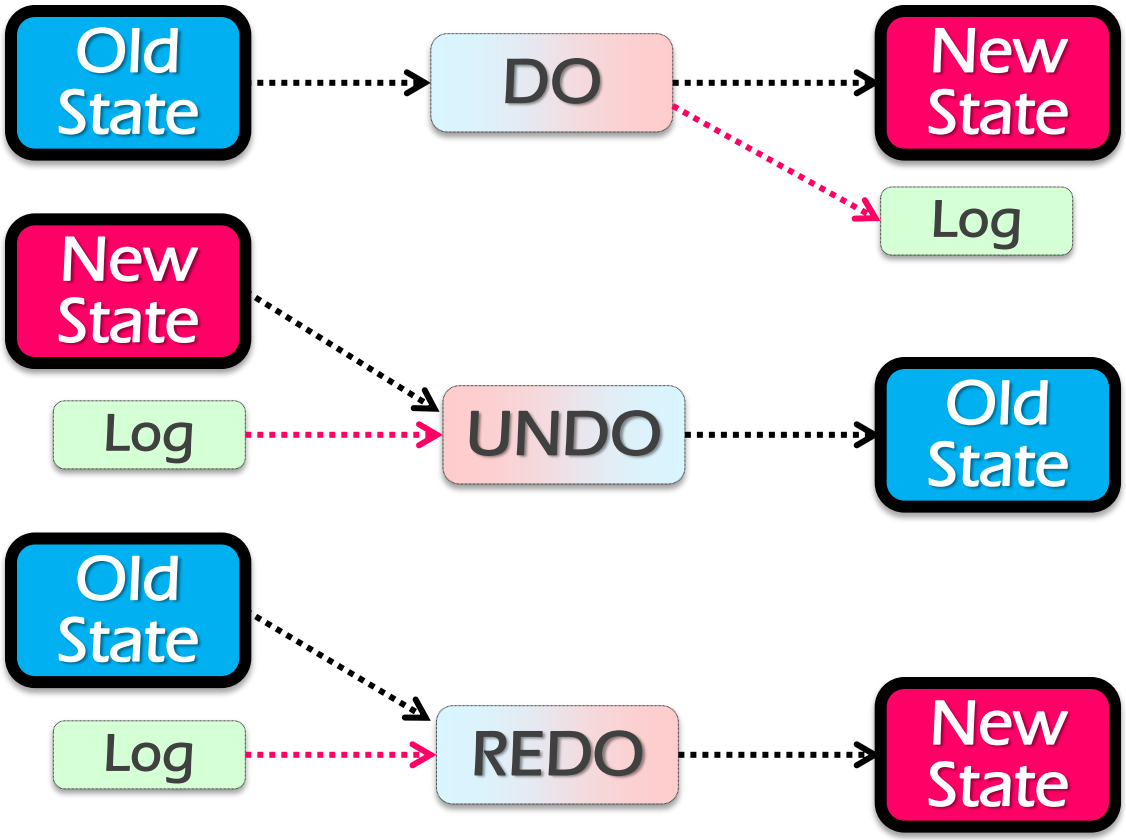
- A set of operations either all finish or none at all
- No intermediate state exist upon recovery

All-or-nothing is one of the guarantees offered by database transactions

Solution: Logging

Keep a log of all update actions Log

Each action has 3 required operations



Logging

Why records both **UNDO** and **REDO** logs

- A transaction might be very long
  → Must checkpoint w/ ongoing transactions
- A long transaction might be aborted
  → Must UNDO abort state when recovery

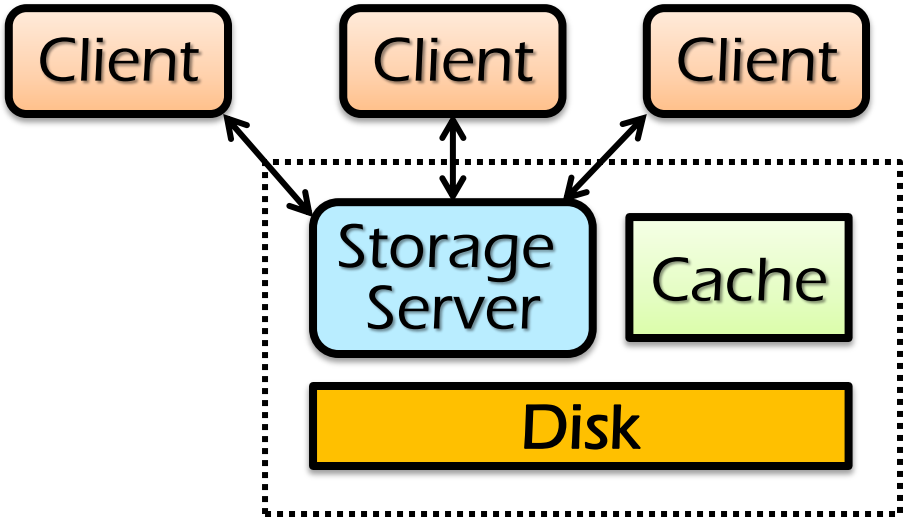
Why we need **checkpoint**

- Avoid aborting long transaction
- No need to recovery from a blank state
- . . .

Concurrency Control

Concurrent transactions

- Multiple application clients are concurrently accessing a storage system
- Transactions might interfere
- Similar to data race in parallel programs



Two-phase Locking

Two-phase locking (**2PL**)

- A growing phase in which the transaction is acquiring locks
- A shrinking phase in which locks are released

In practice

- The growing phase is the entire transaction
- The shrinking phase is at the **COMMIT** time

Optimization

- Use read/write locks instead of exclusive locks

Multi-version Concurrency Control

No locking during transaction execution

Each data item has multiple versions

Multi-version transactions:

- Buffer **WRITE**s during execution
- **READ**s choose the appropriate version
- At **COMMIT** time, system validates if OK to make writes visible (generating **new** versions)

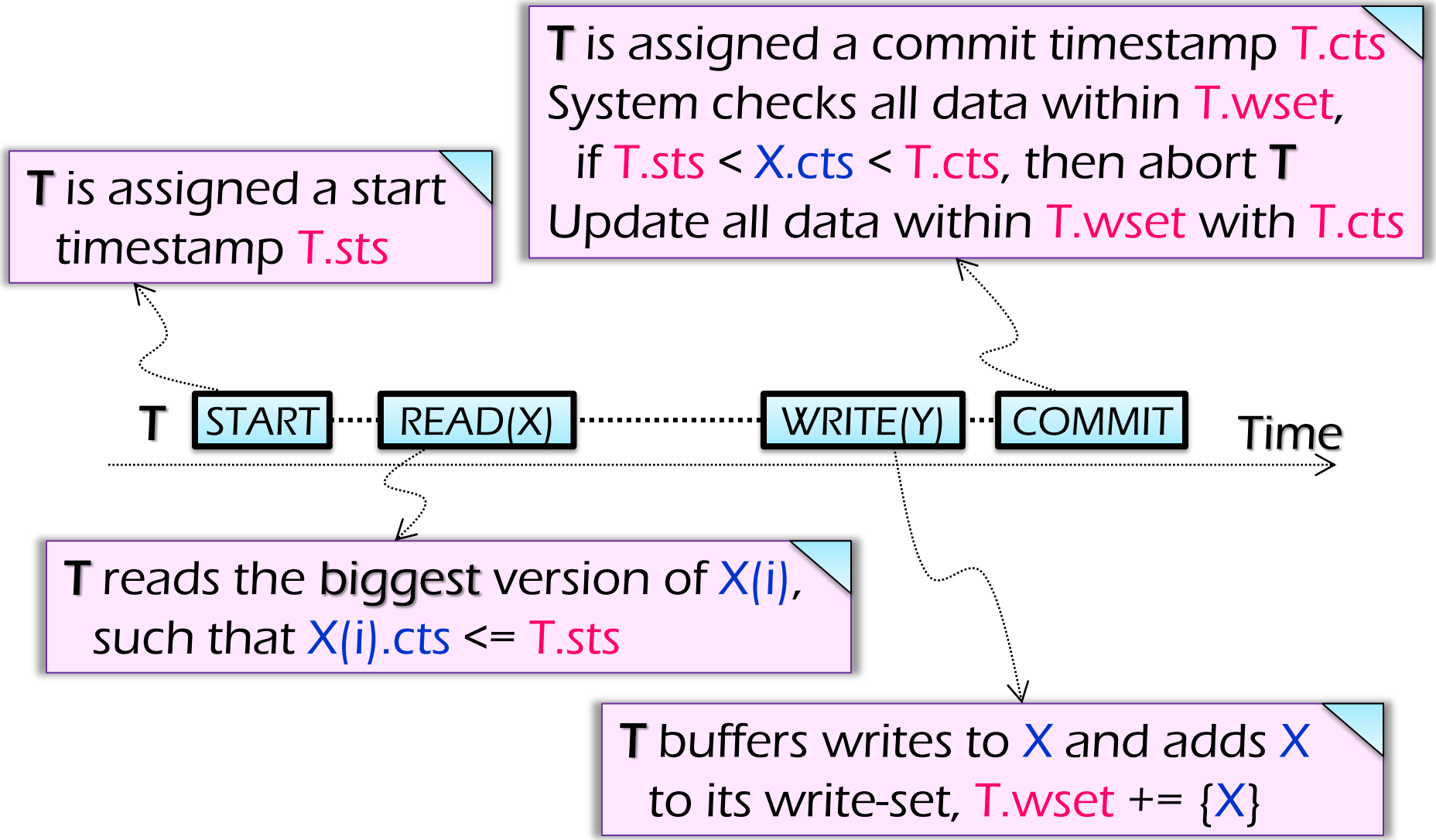
Snapshot Isolation

A popular multi-version concurrency control (MVCC) scheme

Transactions:

- **WRIT**s a lock buffer
- **READ**s a "snapshot" of entire data image
- **COMMIT** only if no write-write conflict

Snapshot Isolation



Two-phase Commit (2PC)

The commit-step itself is two phase

Phase-1: Voting

- Each participant **prepares** to commit, and **votes** on whether or not it can commit

Phase-2: Committing

- Each participant actually **commits** or **aborts**

The Voting Phase

TC asks each **participant** canCommit(T) ?

Participants must prepare to commit using permanent storage before answering YES

- Objects are still locked
- Once a **participant** votes "**YES**"
  → it is not allowed to cause an **ABORT**

Outcome of **T** is uncertain until doCommit(T) or doAbort(T)

- Other **participants** might still cause an **ABORT**

Don't forget about their response after **REBOOT**

The Committing Phase

TC collects all votes

- If unanimous "**YES**", cause **COMMIT**
- If any **participant** voted "**NO**", cause **ABORT**

The fate of the **T** is decided atomically at the **TC**, once all **participants** vote

- **TC** records fate using permanent storage
- Then broadcasts *doCommit(T)* or *doAbort(T)* to **participants**

Don't forget about its decision after **REBOOT**

RSM: Replicated State Machine

RSM is a general replication method

- e.g., apply RSM to lock service

RSM rules

- All replicas start in the same initial state
- Each replica apply ops in the same order
- All ops must be deterministic
- All replicas end up in the same state

RSM: Failure Handling

If primary fails, one backup acts as the **new** primary

Challenges

1. How to reliably detect primary failure?
2. How to ensure no two backups simultaneously become new primary?
3. How to preserve sequential consistency across primary changes?

A fault tolerant distributed consensus protocol

Consensus Goal

Create an algorithm that does not block if a majority of processes are working

Goal: agree on one result among a group of participants

- Nodes may fail (may need stable storage)
- Messages: lost, out of order, or duplicated
- If delivered, messages are not corrupted

The algorithm needs (**2P**+**1**) nodes survive the simultaneous failures of **P** nodes

Paxos

Paxos: fault-tolerant distributed consensus algorithm (the **only known**)

- All nodes agree on the same value despite node failure, network failure and delays

General Approach

- One **proposer** decides to be the leader
- Leader proposes a value and solicits acceptance from **acceptors** (majority)
- Leader announces result or try again

Paxos Pseudo-code

A node decides to be Leader

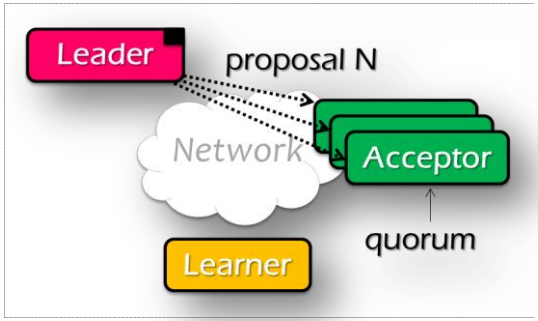
Leader choose $M_n > N_h$

Leader sends <proposal, M_n> to all nodes

Acceptor receives <proposal, N>

```
if N < Nh
    reply <promise-reject>
else
    Nh = N
    reply <promise-ok, Na, Va>
```

Ignore all proposals < N_h



Paxos Pseudo-code

If Leader gets promise-ok from a majority

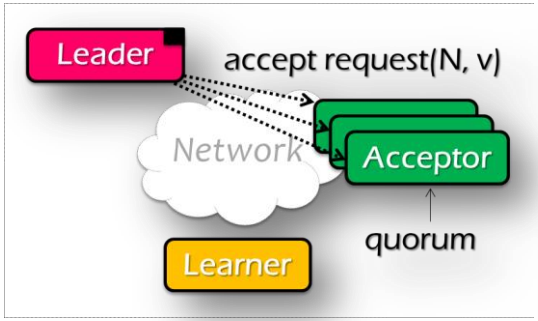
V = non-empty value (highest $\mathbf{N_a}$ received)
if **V = null**, then Leader can pick any **V**
send <accept, $\mathbf{M_n}$, **V**> to all nodes

If Leader fails to get majority promise-ok

delay and restart Paxos

Upon receiving <accept, **N**, **V**>

if $\mathbf{N < N_h}$
    reply <accept-reject>
else
    $\mathbf{N_a = N;\ V_a = V;\ N_h = N;}$
    reply <accept-ok>



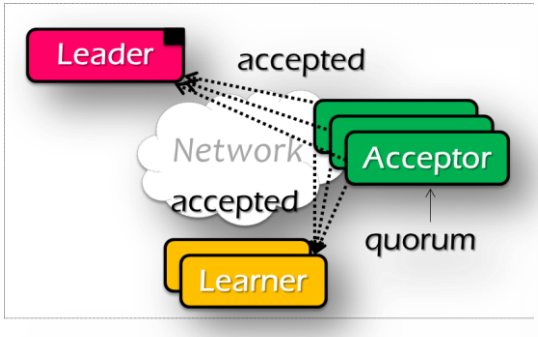
Paxos Pseudo-code

If Leader gets accept-ok from a majority

send <decide, V_a> to all nodes

If Leader fails to get majority accept-ok

delay and restart Paxos



Byzantine Faults

Nodes fail **arbitrarily**

- Failed node performs **incorrect** computation
- Failed nodes **collude**

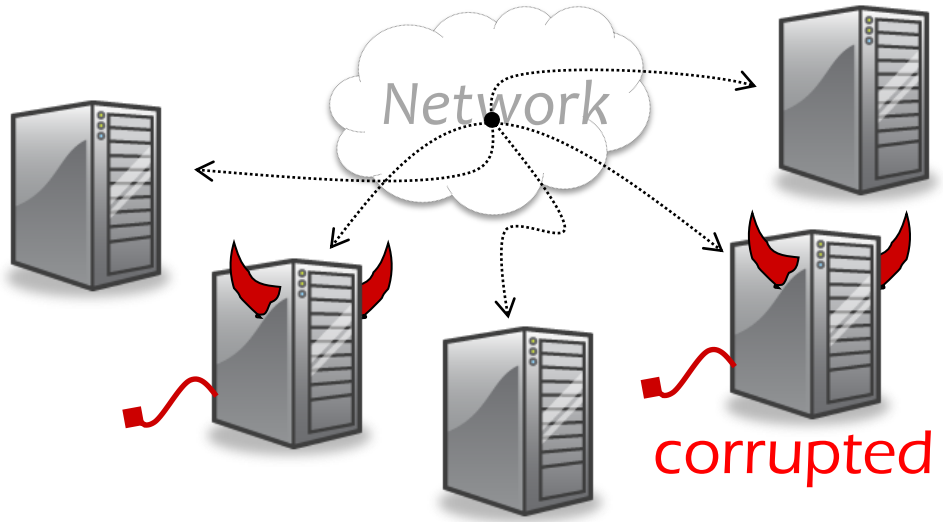
Causes: **attacks**, soft/hardware **errors**

Examples:

- Client asks bank to **deposit** $100, but a *Byzantine* server **subtracts** $100
- Client asks file system to store f1="**aaa**", but a *Byzantine* server returns f1="**bbb**"

Byzantine in DS

A thousand years later...



- N Computers
- Some misbehaviors
- Message passing

HW Fault, SW bug, Security Attack, Misconfiguration

Goals

- All correct nodes share the same global info
- Ensure that N corrupted nodes cannot change the shared global info and maximize N

PBFT Main Ideas

Static configuration (3f+1 nodes across views)

To deal with loss of agreement

- Bigger quorum (2f+1 out of 3f+1 nodes)

To deal with malicious primary

- 3-phase protocol to agree on sequence number

Need to authenticate communications

PBFT Strategy

Primary runs the protocol in the normal ops

- Client sends **signed** request to primary or ALL

Replicas watch the primary and do a view change if it fails (or faulty)

- Pick primary in turn ⟶ *avoid dictatorship*
- At least f+1 replicas to push view change

avoid anarchy

Distributed Computation

Assumption for Data-parallel

No dependency among data

Data can be split into equal-size chunks

Each process can work on a chunk

Master/worker approach

- Master
  1. Initialize array and splits it according to # of workers
  2. Send each worker the sub-array
  3. Receive the results from each worker
- Worker
  1. Receive a sub-array from master
  2. Perform processing
  3. Send results to master

MapReduce Programming Model

Allows user to process *huge* amounts of data on *thousands* of nodes

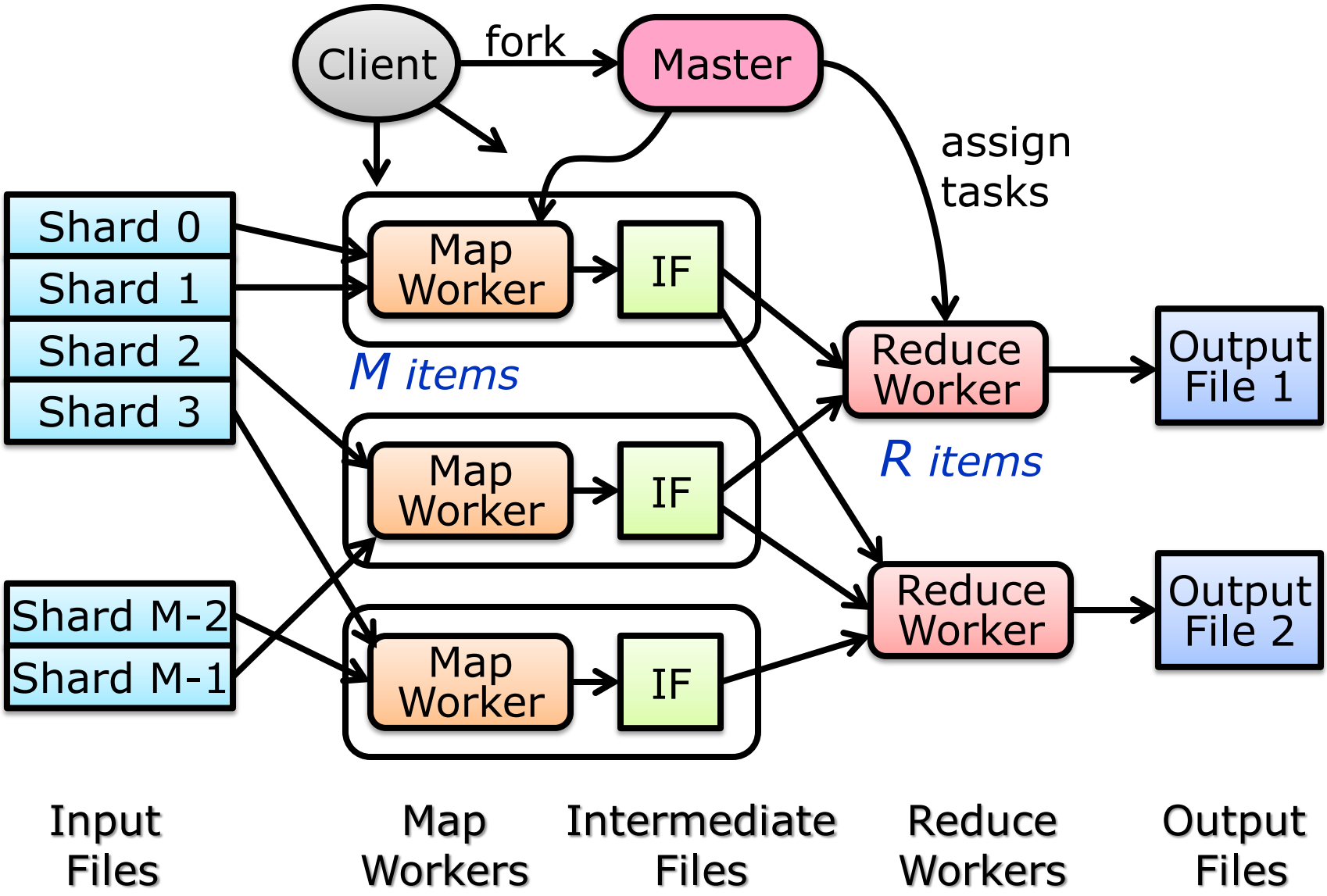
Framework for data-parallel computing

Programmers get **simple** API

Don't have to **worry** about handling

- ~~Parallelization~~
- ~~Data distribution~~
- ~~Load balancing~~
- ~~Fault tolerance~~

The Complete Picture



Issues

Locality

Fault tolerance

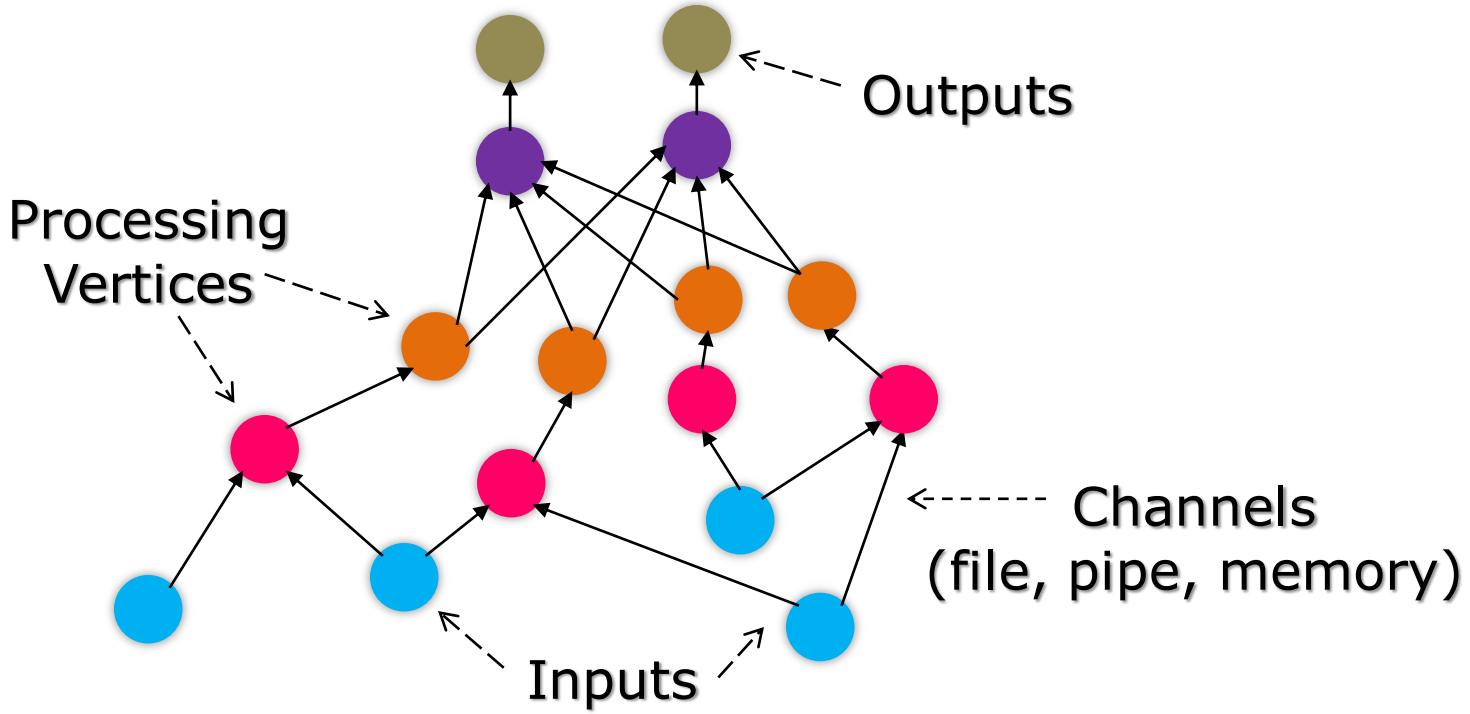
Straggler

. . .

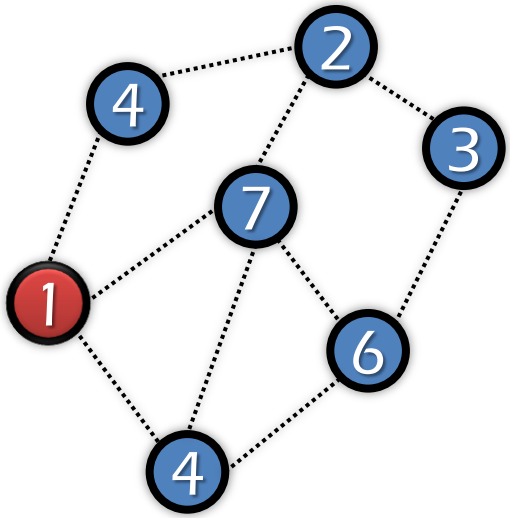
Dryad

Computations expressed as a graph

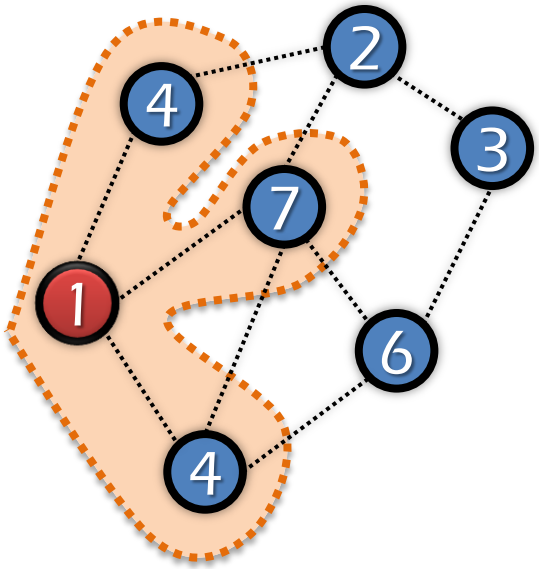
- Vertices are computations
- Edges are communication channels



Graph Parallel Algorithm



Dependency
Graph



Local
Updates

Iterative
Computation

Why not MapReduce?

Difficult to express dependent data in MR

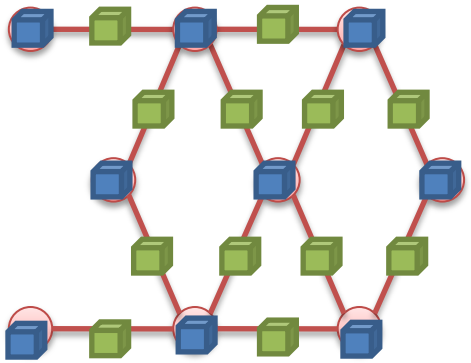
- Substantial data transformations
- User managed graph structure
- Costly data replication

MR runtime is not optimized for iteration

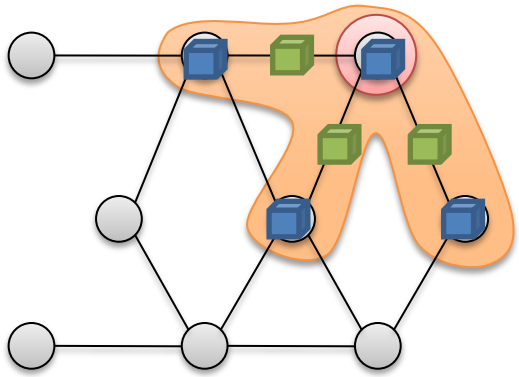
- Persistent I/O (e.g., GFS)

GraphLab Framework

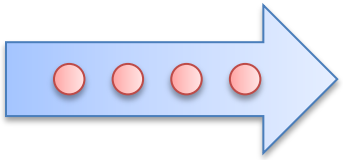
Graph Based
Data Representation



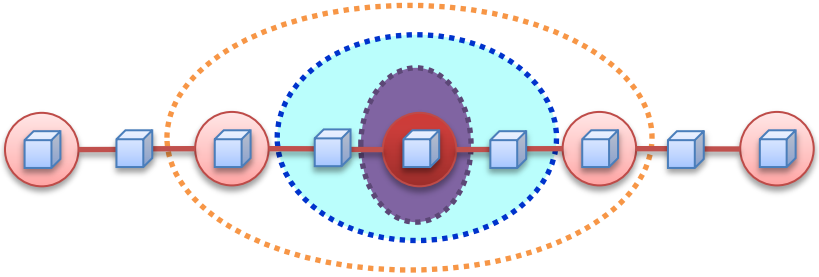
Update Functions
User Computation



Scheduler



Consistency Model



Another Choice: Matrix

Array-based languages (*e.g.*, *R*) are suitable for implementing complex algorithms

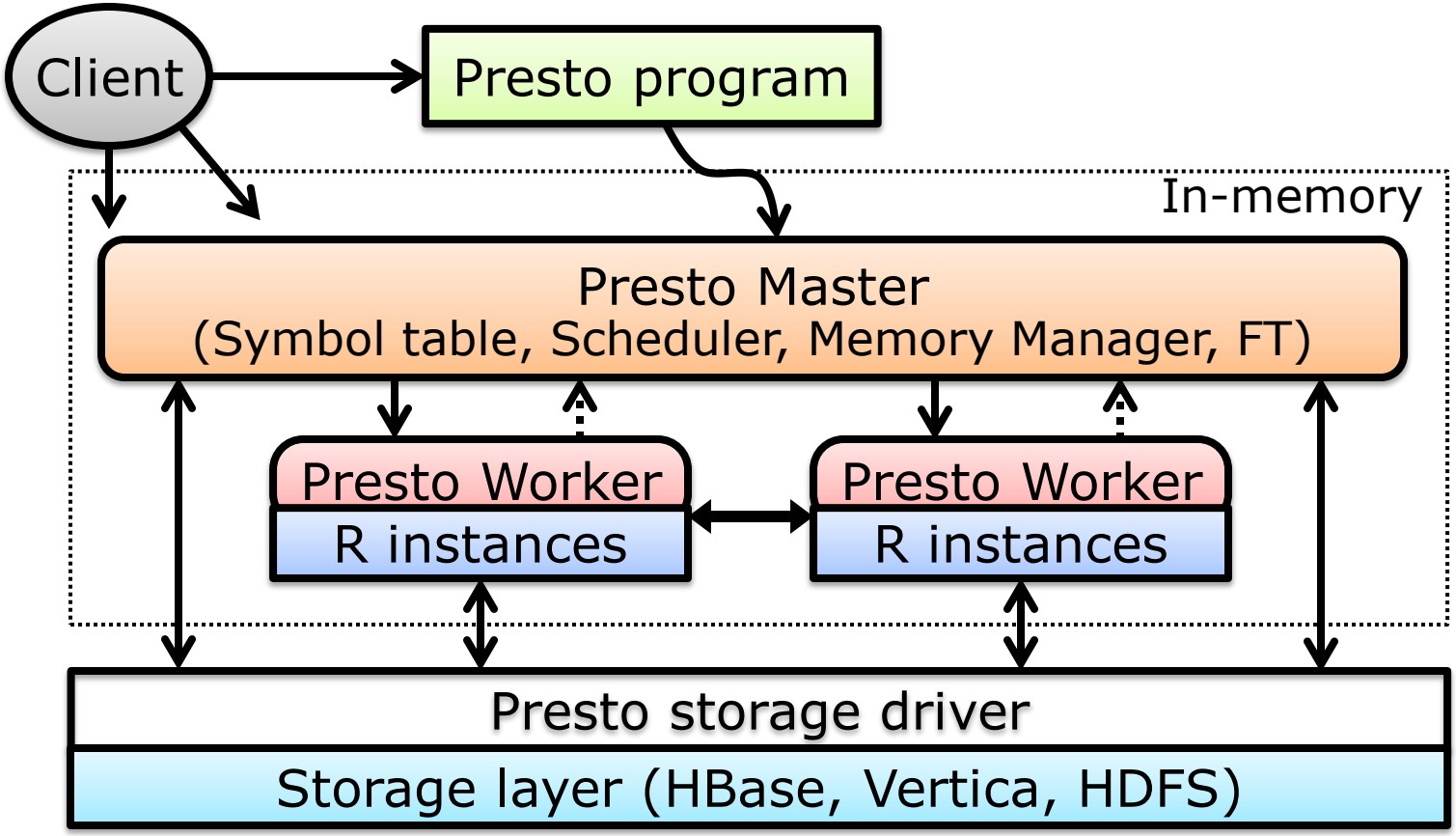
> *Randomness combined with* ***linear algebra*** *is a powerful tool for modern problems.*
>
> *-- Ravi Kannan, FCRC 2011*

R provides an interactive environment to analyze data

- Conditional execution (***if***), loops (***for***, ***while***, ***repeat***), and uses ***array operators*** written in *C/C++* or *FORTRAN*

Programming Model

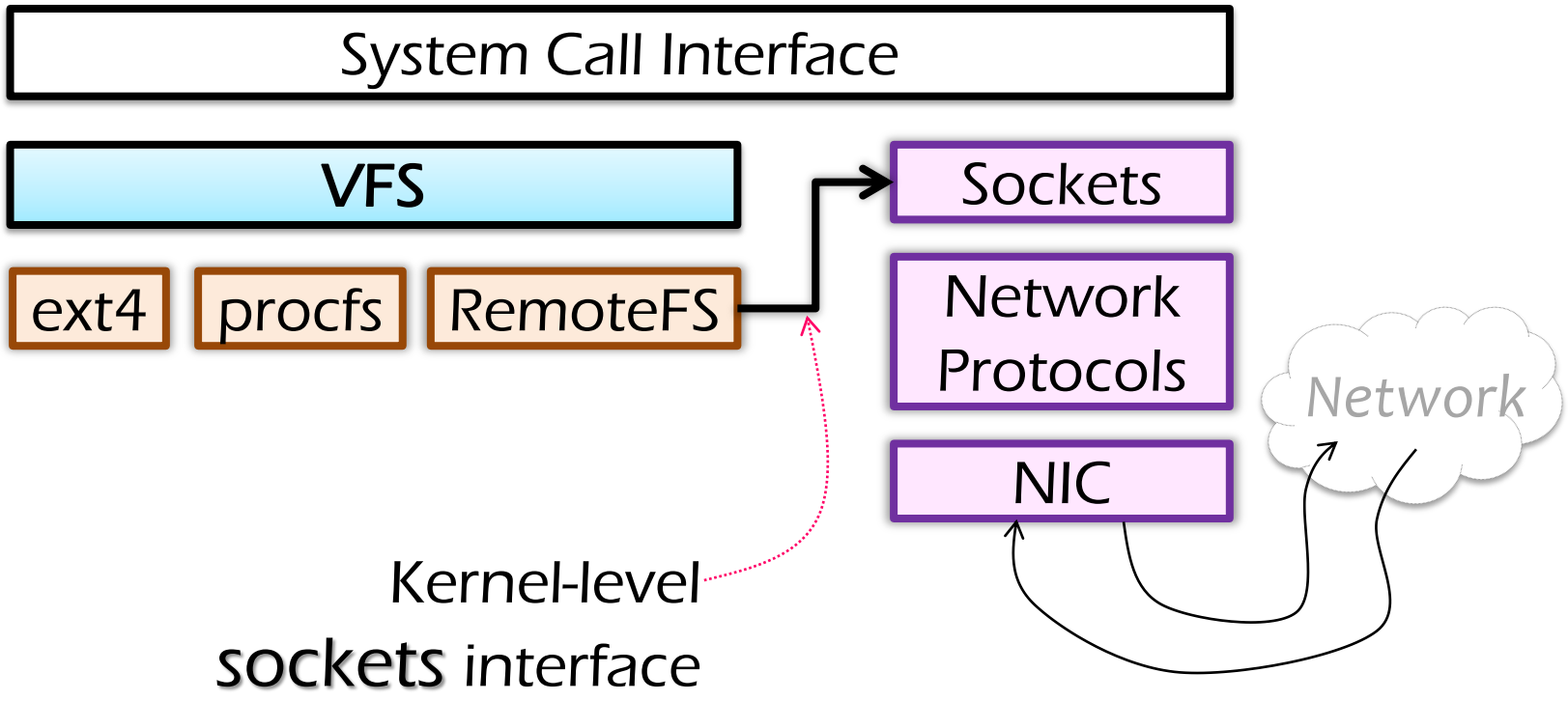
Presto = ***R*** + (***darry, foreach, spilts, update***)



Distributed Storage

Accessing Remote Files

Implement the client module as a FS under VFS



NFS: Network File System

Design Goals

- Any machine can be a client or server
- Must support diskless workstations
- Heterogeneous system must be supported
  - Different HW, OS, underlying file system
- Access transparency
- Recovery from failure
  - Stateless, UDP, client retries
- High performance
  - Use caching and read-ahead

GFS Goals

Scalable distributed file system

Designed for large data-intensive applications

Fault-tolerant; runs on commodity hardware

Delivers high performance to a large number of clients

Design Issues

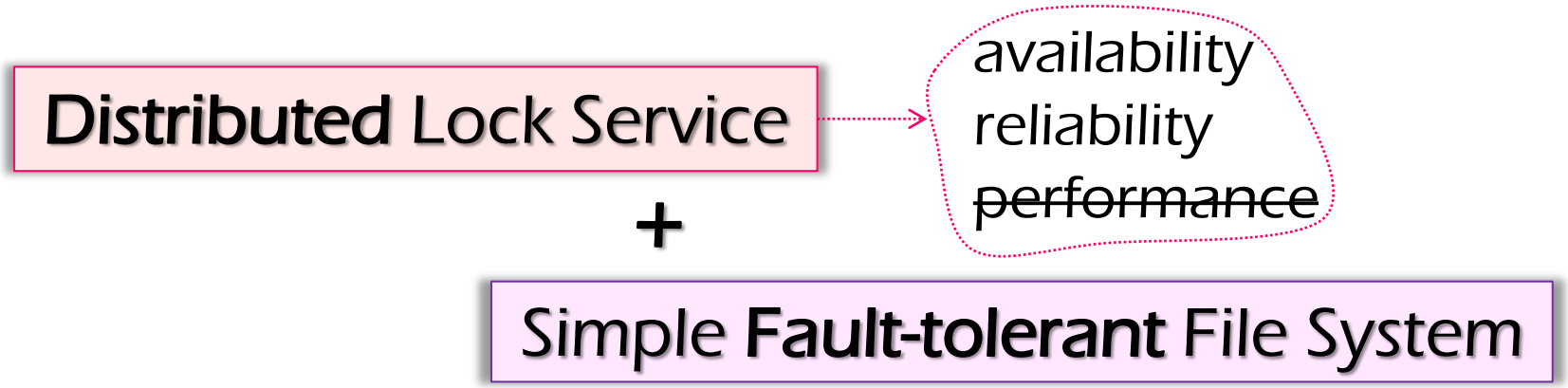
Large Chunks

One Master

Read/Write ops

. . .

Chubby



Interfaces:

- File access / Event Notification / File locking

Used for:

- Coarse-grained, long-term locks
  (Tips: hours or days, not seconds)
- Storing small amounts of data
  (e.g., system config info, primary identification)

Tentative Grading

Exam (40%)

Topic Quiz (10%)

- 1 paper before lecture, and hand in answers to paper questions before lecture

Team Show (50%)

- Presentation (20%)
- Demo (20%)
- Ask Questions (10%)

Score

ID	Show	Question	Quiz	ID	Show	Question	Quiz
0120379003	33.4	10	8	1120379046	35	10	10
1120379002	35.8	8	10	1120379047	35	5	6
1120379008	33.6	5	10	1120379048	35	8	5
1120379010	35.6	8	10	1120379051	35	5	9
1120379011	36.8	9	10	1120379055	35	8	10
1120379013	35.6	0	10	1120379057	35.6	9	10
1120379014	36.8	10	10	1120379059	35.8	10	10
1120379019	35	10	10	1120379061	33.4	0	3
1120379020	35	10	10	1120379066	33.6	5	9
1120379023	35.8	5	10	1120379069	35.6	5	10
1120379027	33.4	10	10	1120379077	34.6	10	10
1120379028	35.6	10	10	1120379080	34.6	10	10
1120379036	35.6	10	10	1120379085	34.6	5	9
1120379040	35	10	10	1120379088	36.8	10	10
1120379045	35	0	10	1120379090	33.6	10	10

Final Exam

Location: 陈瑞球楼 203

Data: 2013.06.20

Time: 14:00 ~ 16:00

THANKS