Lecture 28: Distributed Systems

CS 105

May 6, 2019

Why not just use one computer?

- computers fail
- limited resources
- physical location
- nonuniform hardware

What is a distributed system?

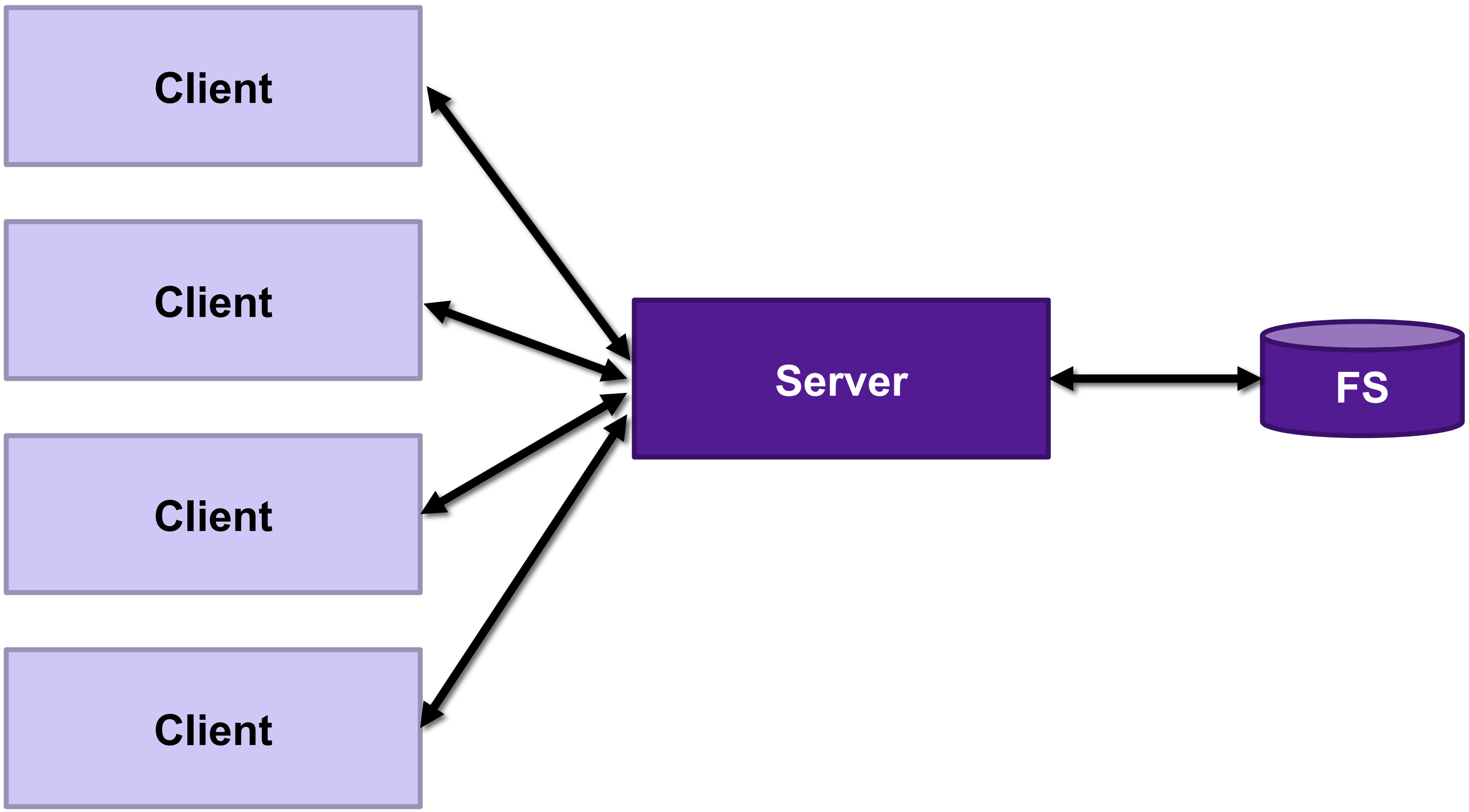
- A **distributed system** is a collection of autonomous computing elements that appears to its users as a single, coherent system

- A **distributed system** is several computers doing something together. Thus, a distributed system has three primary characteristics: multiple computers, interconnections, and shared state.

Properties we want

- **Transparency:** Hide that resource is physically distributed across multiple computers
- **Reliability:** system doesn't go down/go wrong when component(s) fail
- **Consistency:** appears as all one system
- **Scalability:** can grow (add more nodes, memory, etc.)

Example: a networked file system



Communication

- Option 1: socket-based communication
- Option 2: remote procedure calls

Remote Procedure Calls

- RPCs are a type of client/server communication
- attempts to make remote procedure calls look like local procedure calls

```
{ …
  foo();
}

void foo(){
  invoke_remote_foo();
}
```

Problems with RPCs

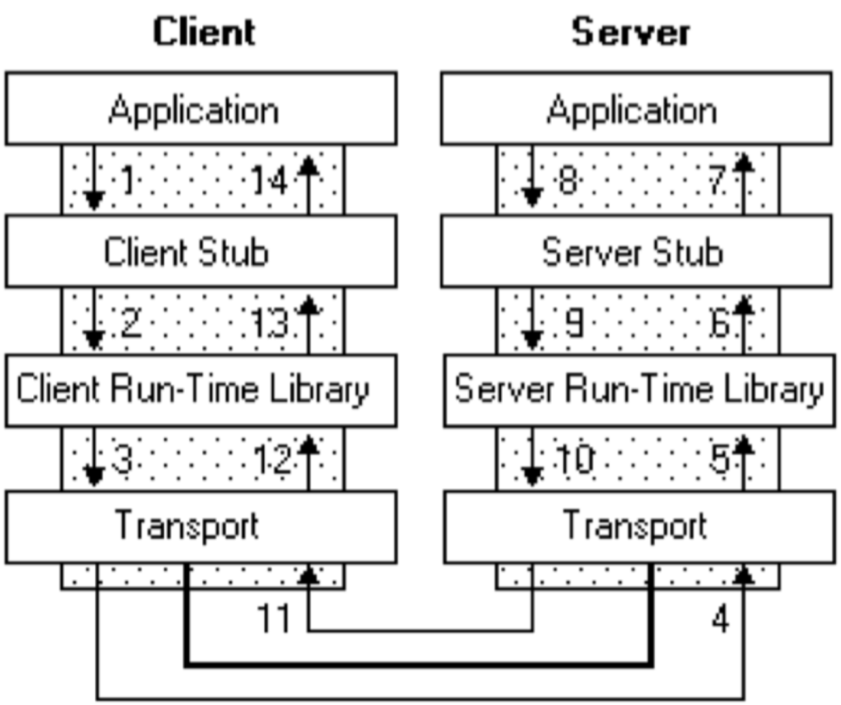
- Heterogeneity
  - Client needs to rendezvous with the server
  - Server must dispatch to the required function
  - Different address spaces, data representation

- Failure
  - What if messages get dropped?
  - What if client, server, or network fails?

- Performance
  - Procedure call takes ≈ 10 cycles ≈ 3 ns
  - RPC in a data center takes ≈ 10 μs (10^3 times slower)
  - In the wide area, typically 10^6 times slower

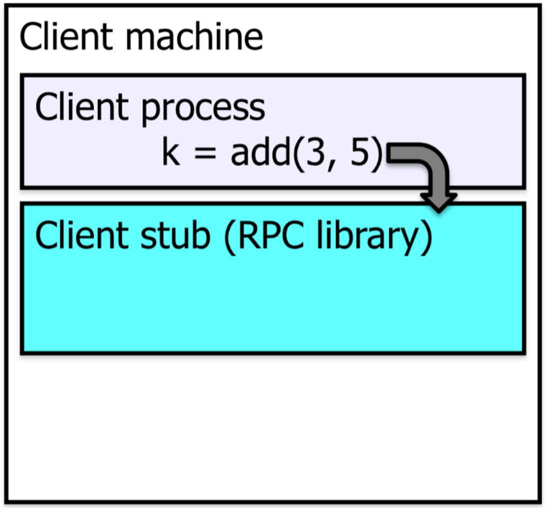
Stubs

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - **Marshals** arguments into machine -independent format
  - Sends request to server
  - Waits for response
  - **Unmarshals** result and returns to caller
- Server stub
  - **Unmarshals** arguments and builds stack frame
  - Calls procedure
  - Server stub **marshals** results and sends reply



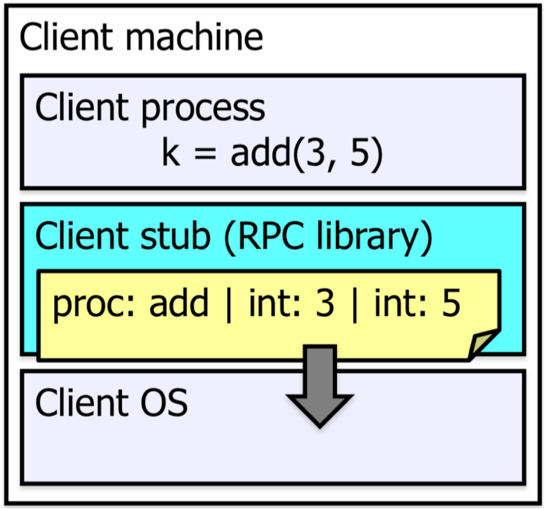
Using RPCs

1. Client calls stub function (pushes parameters onto stack)



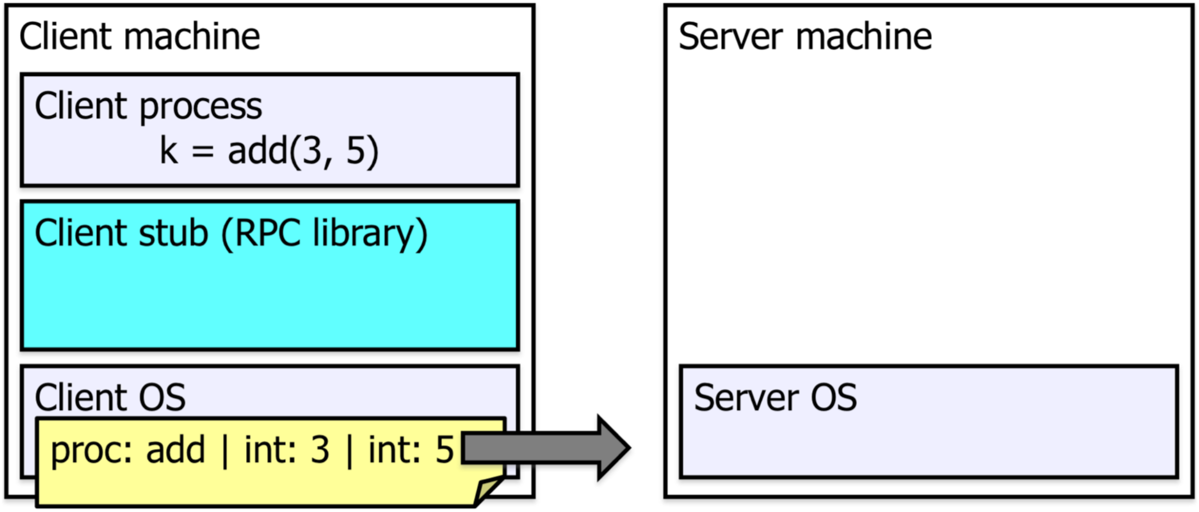
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message



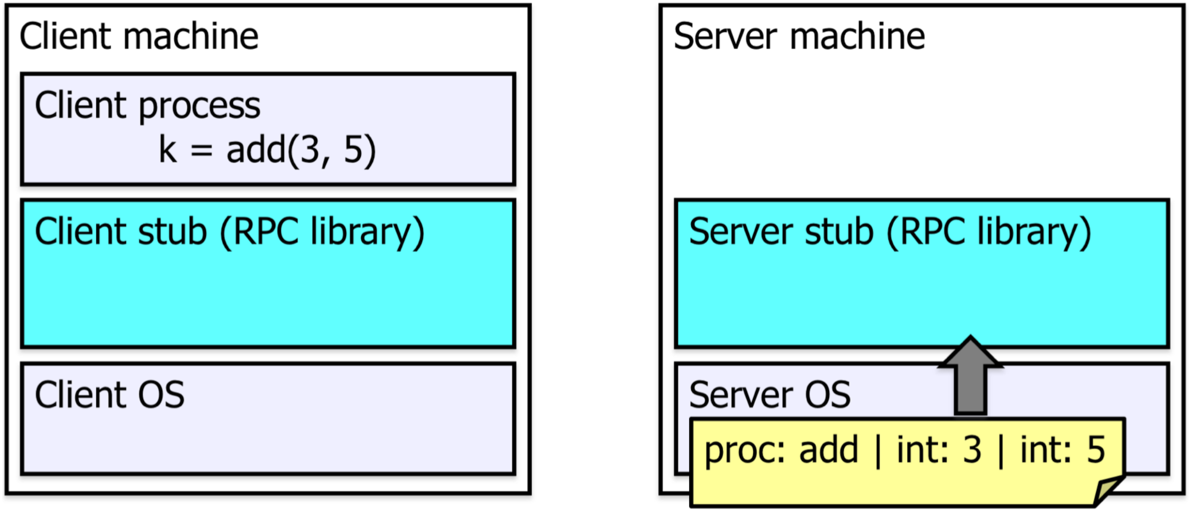
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server



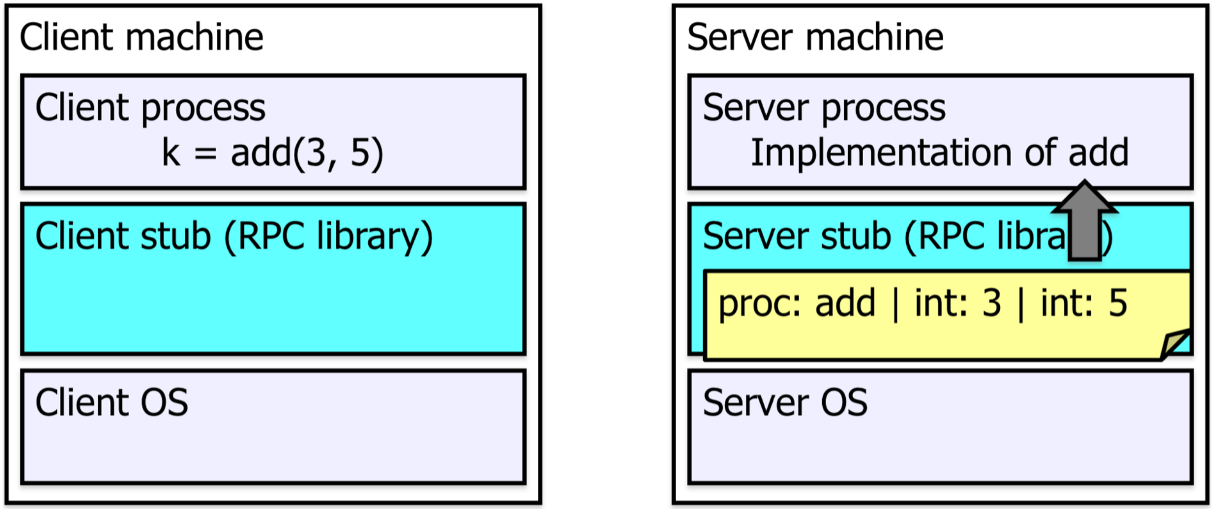
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub



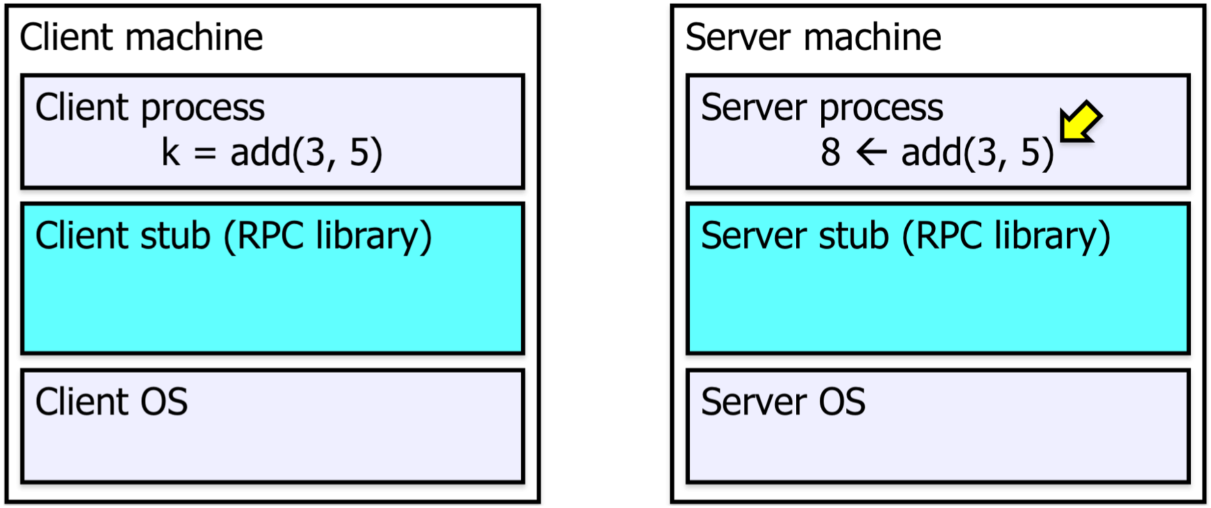
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function



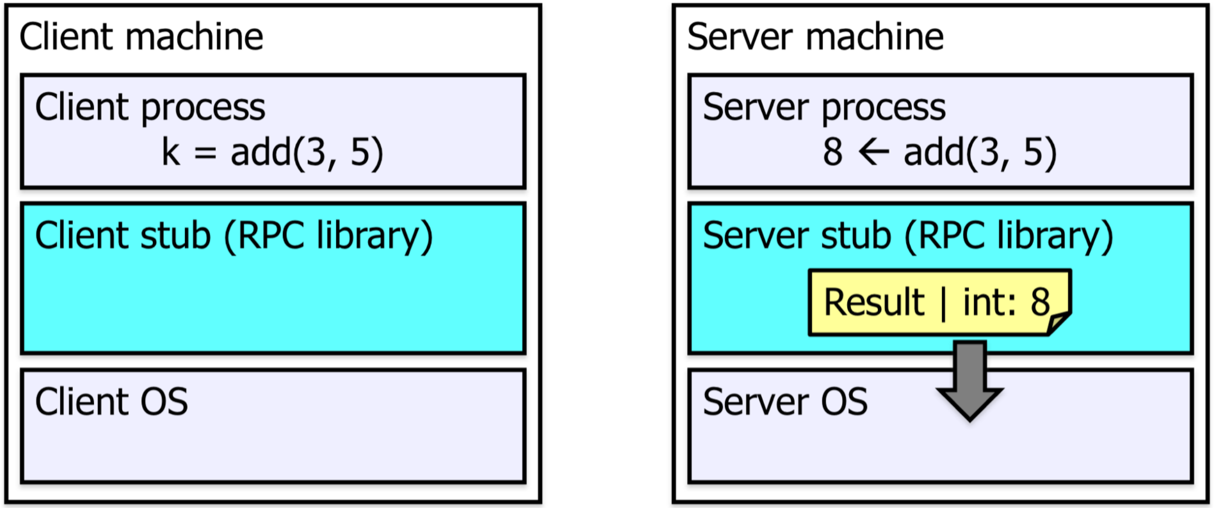
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function
6. Server function runs, returns a value



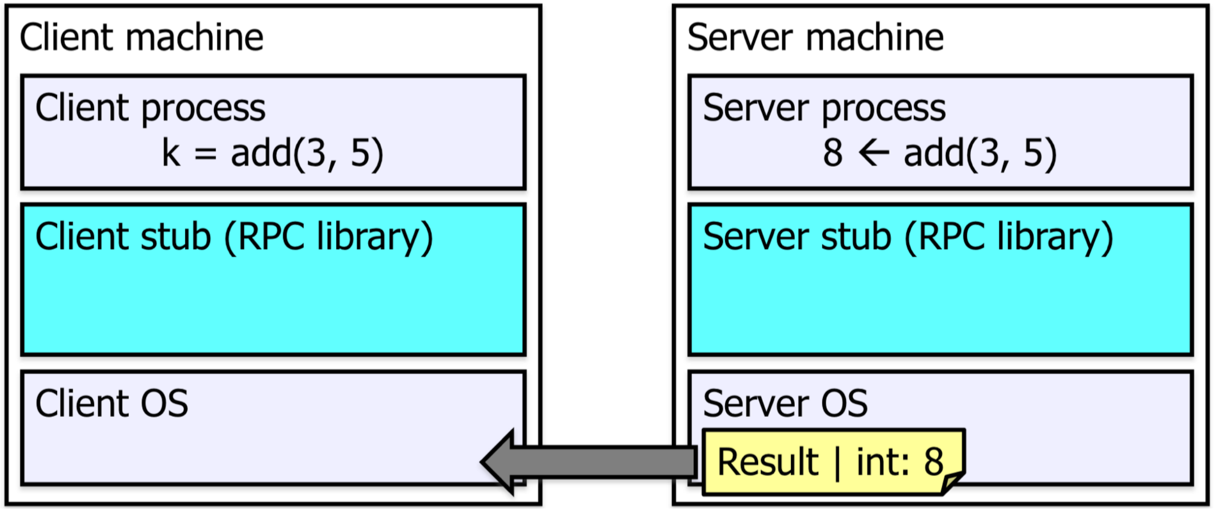
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function
6. Server function runs, returns a value
7. Server stub marshals the return value, sends msg



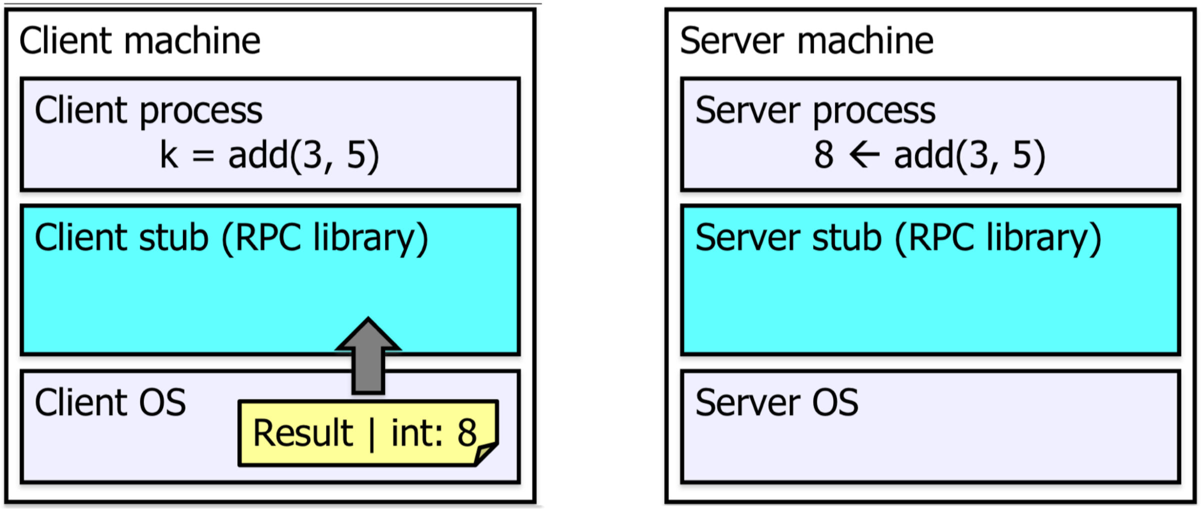
Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function
6. Server function runs, returns a value
7. Server stub marshals the return value, sends msg
8. Server OS sends the reply back across the network



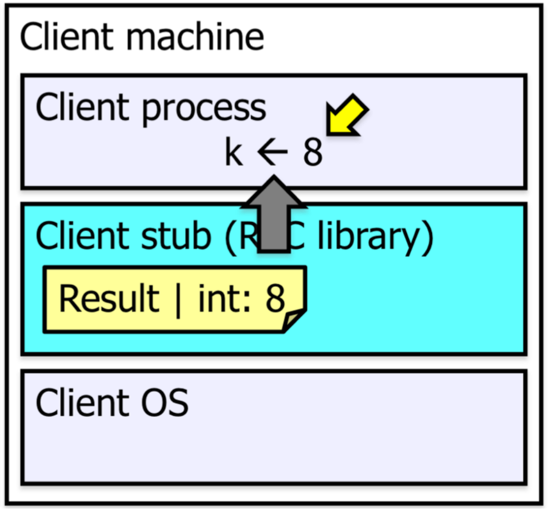
Using RPCs

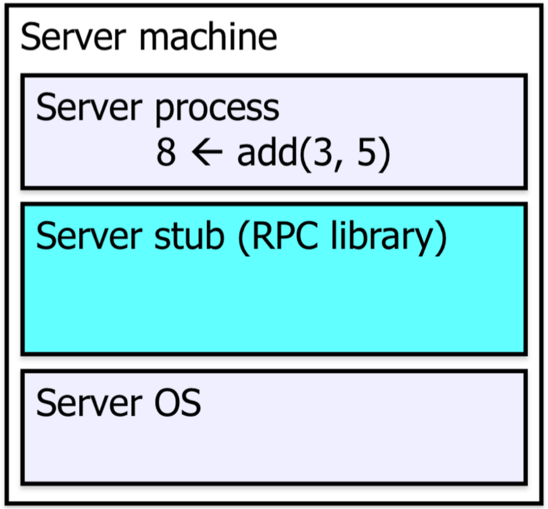
1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function
6. Server function runs, returns a value
7. Server stub marshals the return value, sends msg
8. Server OS sends the reply back across the network
9. Client OS receives the reply and passes up to stub



Using RPCs

1. Client calls stub function (pushes parameters onto stack)
2. Stub marshals parameters to a network message
3. OS sends a network message to the server
4. Server OS receives message, sends it up to stub
5. Server stub unmarshals parameters, calls server function
6. Server function runs, returns a value
7. Server stub marshals the return value, sends msg
8. Server OS sends the reply back across the network
9. Client OS receives the reply and passes up to stub
10. Client stub unmarshals return value, returns to client





RPC Failures

- Request from client to server lost
- Reply from server to client lost
- Server crashes after receiving request
- Client crashes after sending request

look the same to client (refers to the first three items)

RPC Failures

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

Bad solution: replicate local behavior

- Make remote behavior identical to local behavior: every partial failure results in complete failure
  - Abort and reboot the whole system
  - Wait patiently until system is repaired

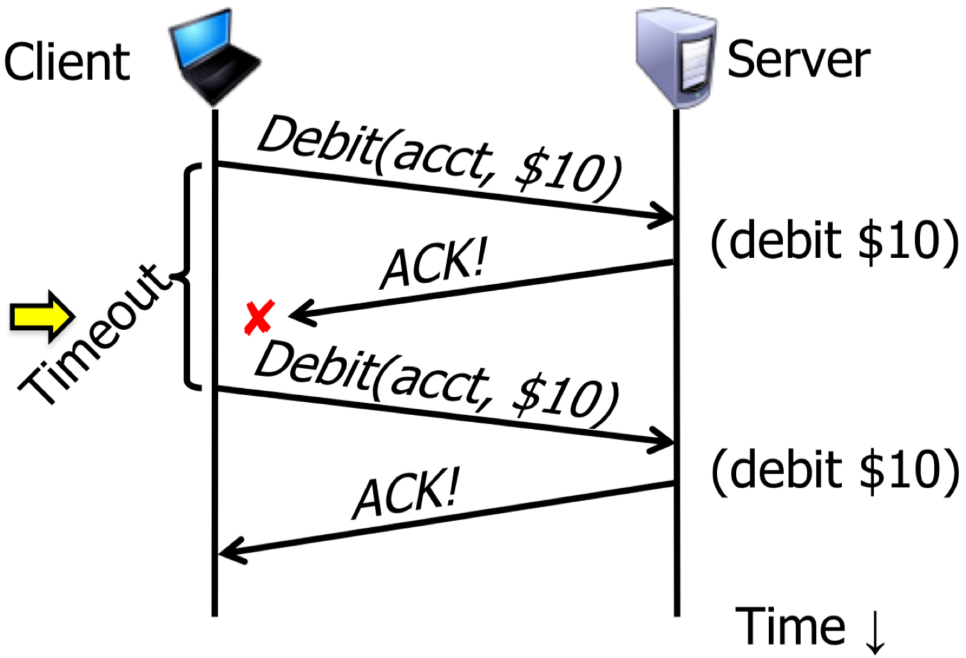
- Problems with this solution:
  - Many catastrophic failures
  - Clients block for long periods
  - System might not be able to recover

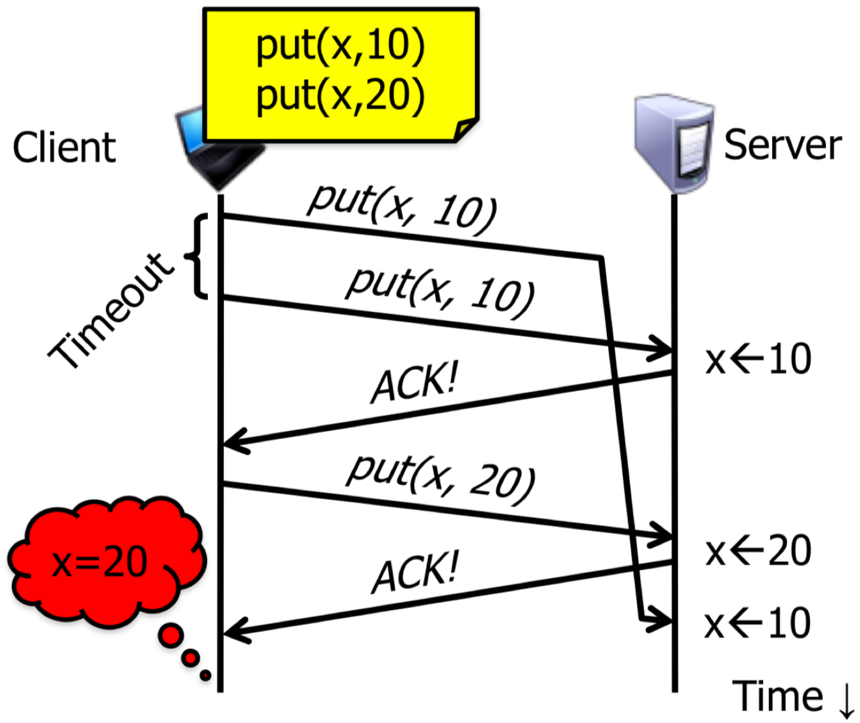
Actual solution: break transparency

- Exactly-once
  - Impossible in practice
- At-least-once
  - Only for idempotent operations
- At-most-once
  - Zero, don’t know, or once
- Zero-or-once
  - Transactional semantics

At-least-once semantics

- Keep retrying on client side until you get a response



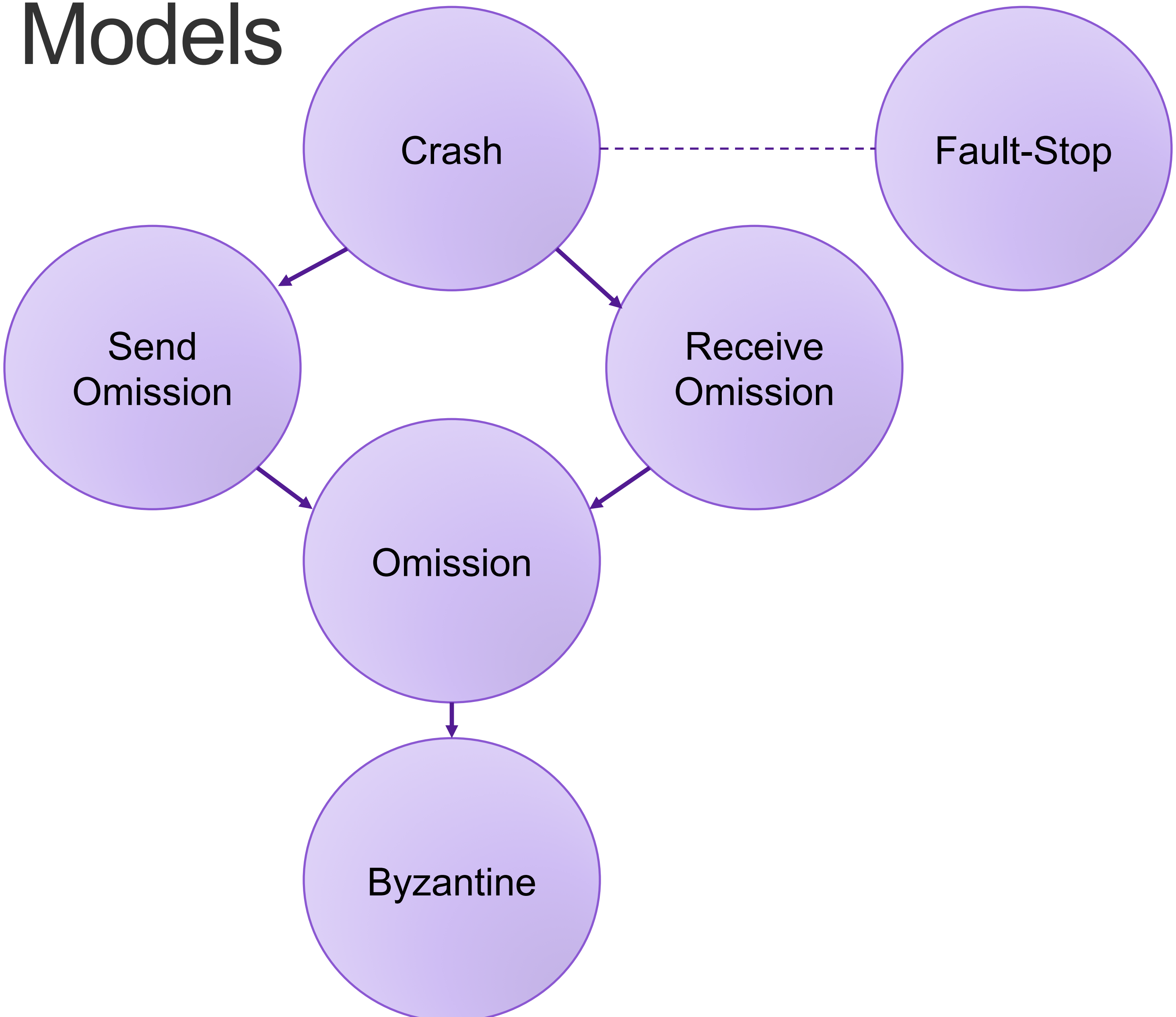


- Ok for idempotent operations
- Ok if application handles duplication/re-ordering

At-most-once semantics

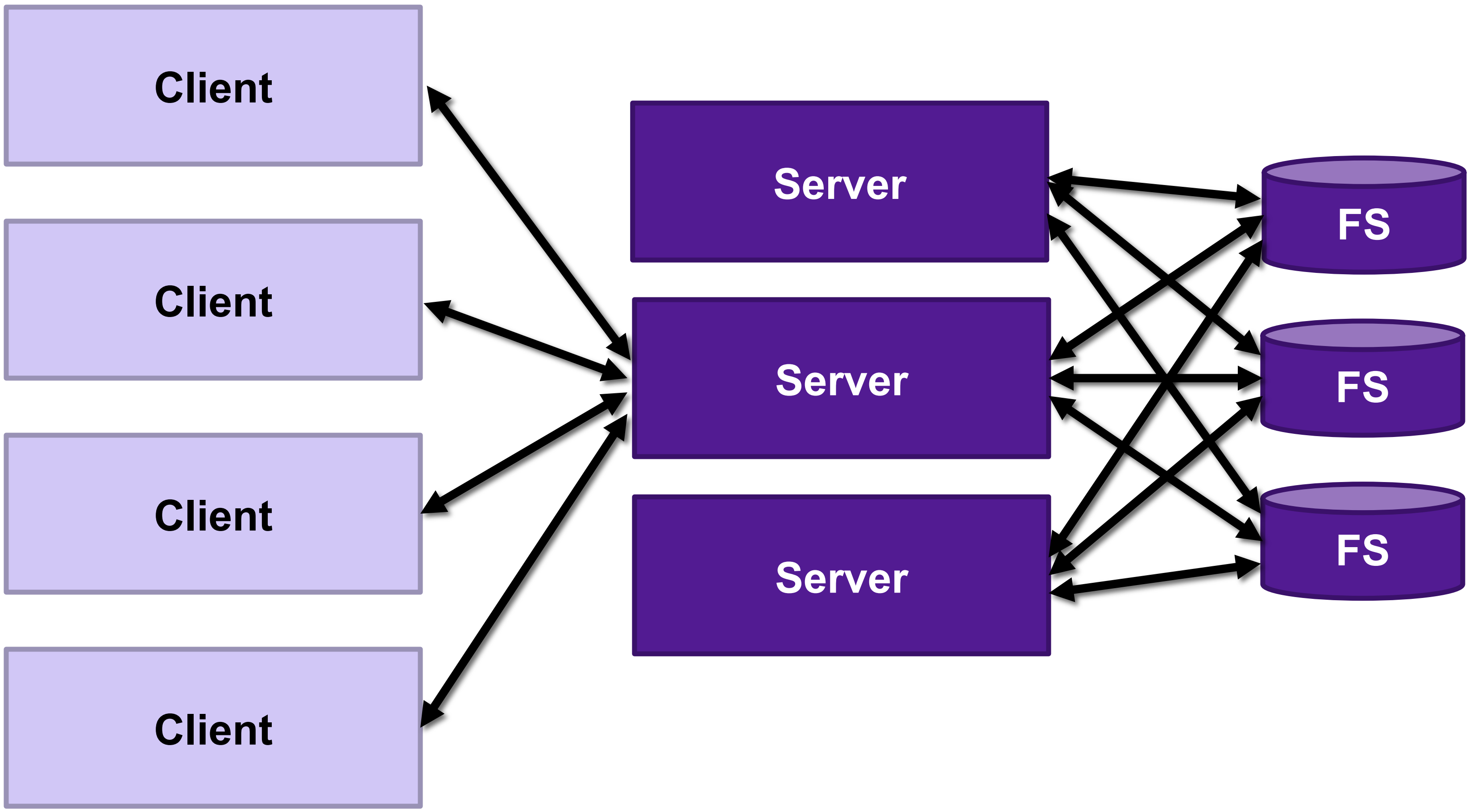
- Server might get same request twice...
- Must re-send previous reply, not process request
  - Implies: keep cache of handled requests/responses
  - Discard replies after client confirmed receipt (how?)
- Must be able to identify requests
  - Same name, same arguments = same request
  - Give each RPC an ID, remember all RPC IDs handled
  - Have client number RPC IDs sequentially, keep sliding window of valid RPC IDs
    - Never re-use IDs! Store on disk, or use boot time, or use big random numbers.

Failure Models

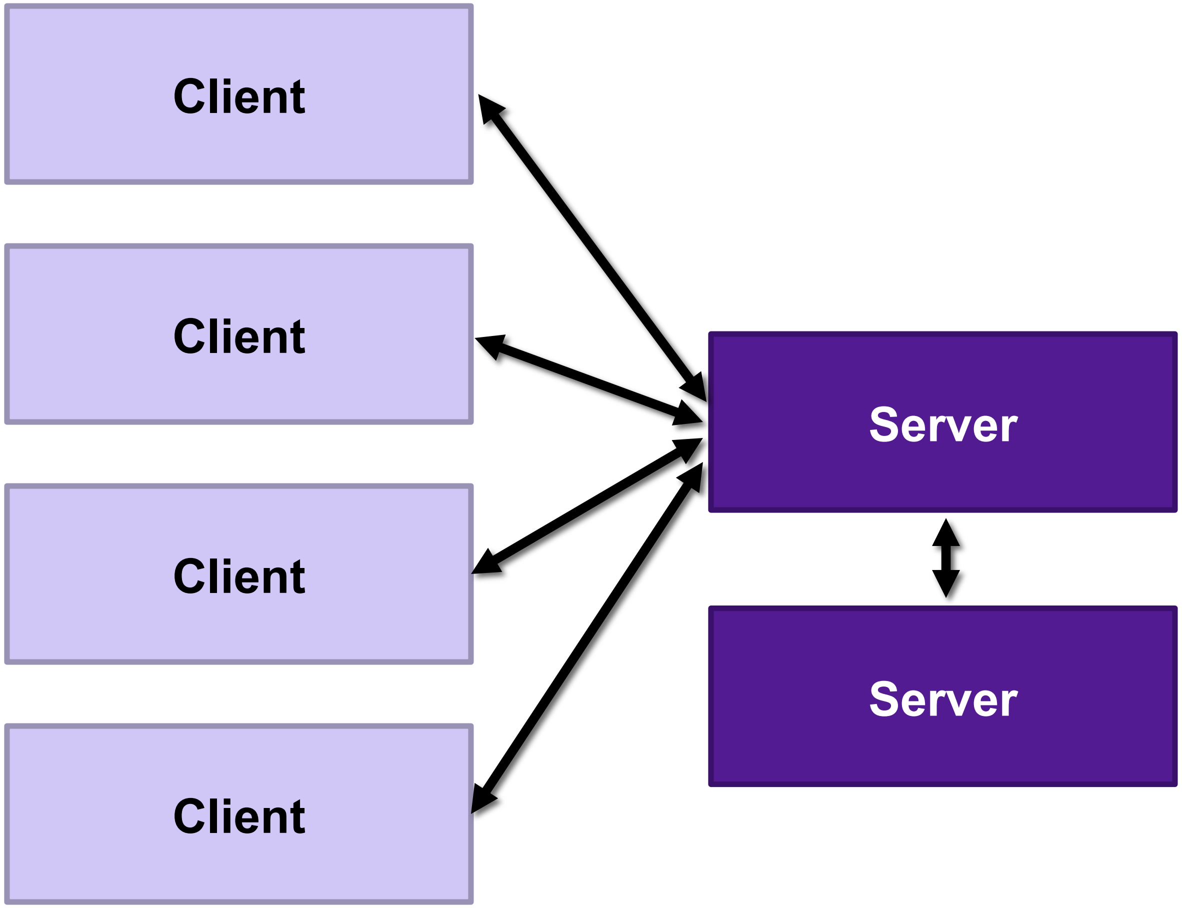


Handling Server Failure

- To tolerate faults, replicate functionality



Primary/Backup



Consensus