What is a Distributed System?

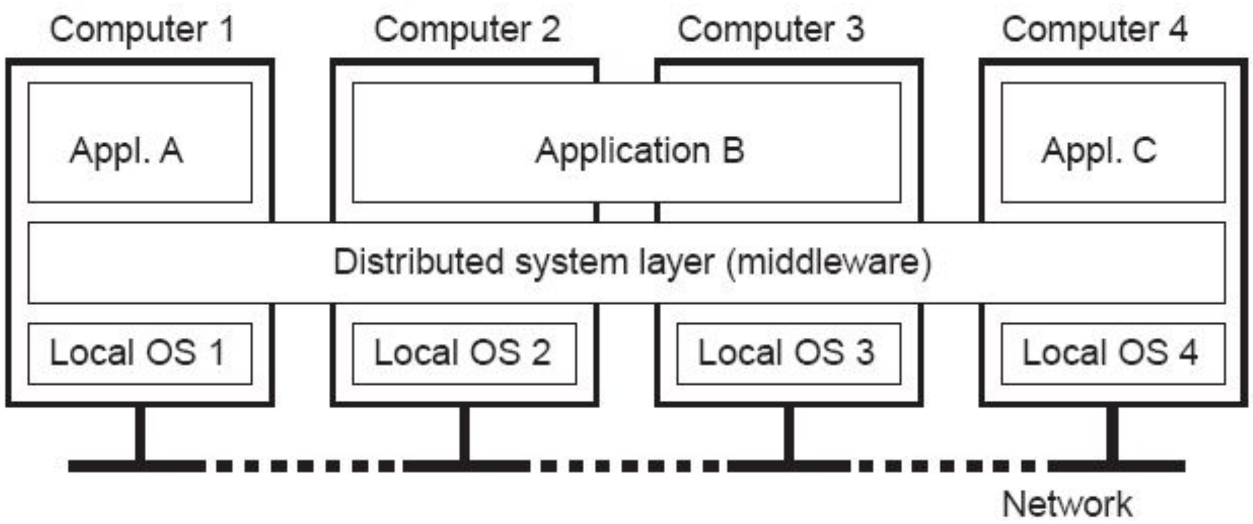
- You know you have one …

  … when the failure of a computer you’ve never heard of stops you from getting any work done

  (*L. Lamport, ‘84*)

What is a Distributed System?

A collection of *independent computers* that appears to its users as a *single coherent system*



- *Independent* hardware installations
- *Uniform software layer* (middleware)

Collection of *independent* components that appears to its users as a *single coherent system*

Requirement: Components need to communicate

- Shared memory
- Message exchange (our focus for this course)

[Detour] Message Passing vs. Shared Memory

Message passing

- Why good? All sharing is explicit – less chance for error
- Why bad? Overhead

  Data copying, across protection domains (context switches)

Shared memory

- Why good? Performance

  Data access w/o crossing protection domains
- Why bad?

  error prone - things change "behind your back"

  more expensive

Enforced modularity!

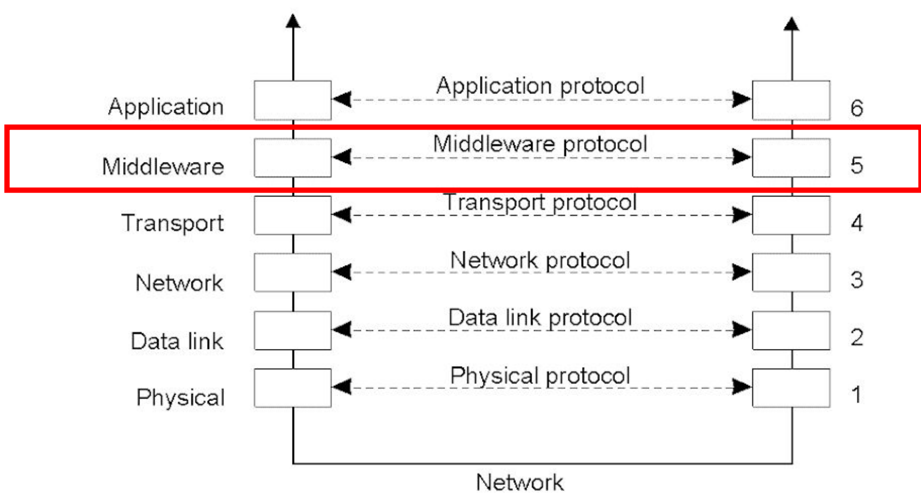
Collection of *independent* components that appears to its users as a *single coherent system*

Requirement: Components need to communicate

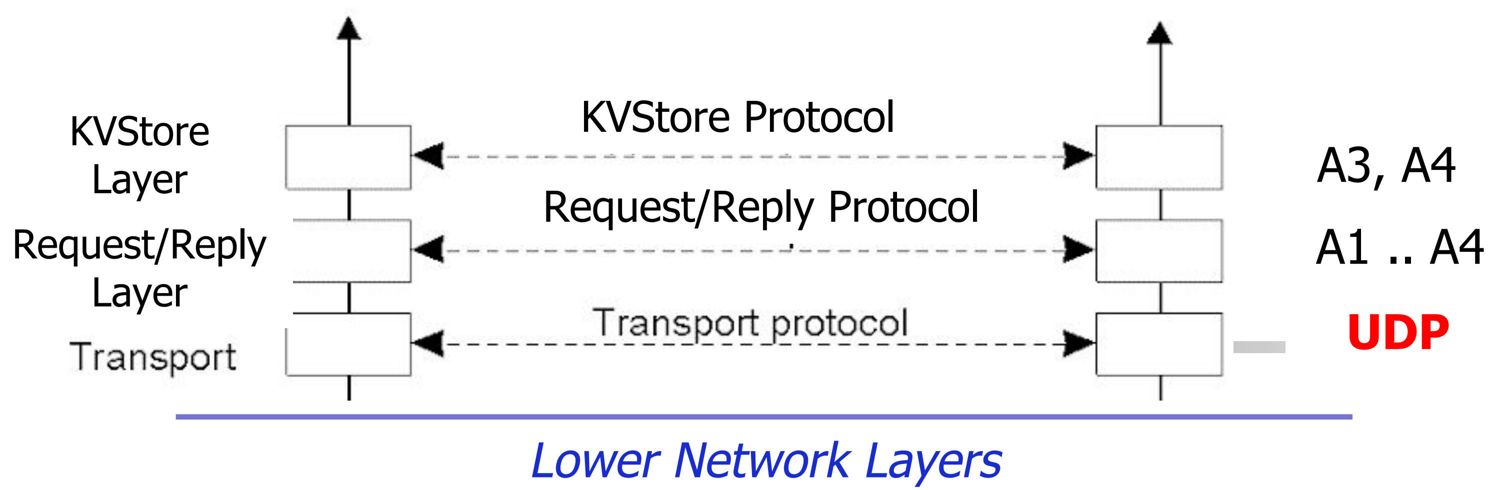
- Shared memory
- Message exchange (our focus for this course)

⇒ need to agree on many things: **Protocols**

- how to send/receive data (i.e. what transport to use),
- data formats,
- fault handling,
- naming, …

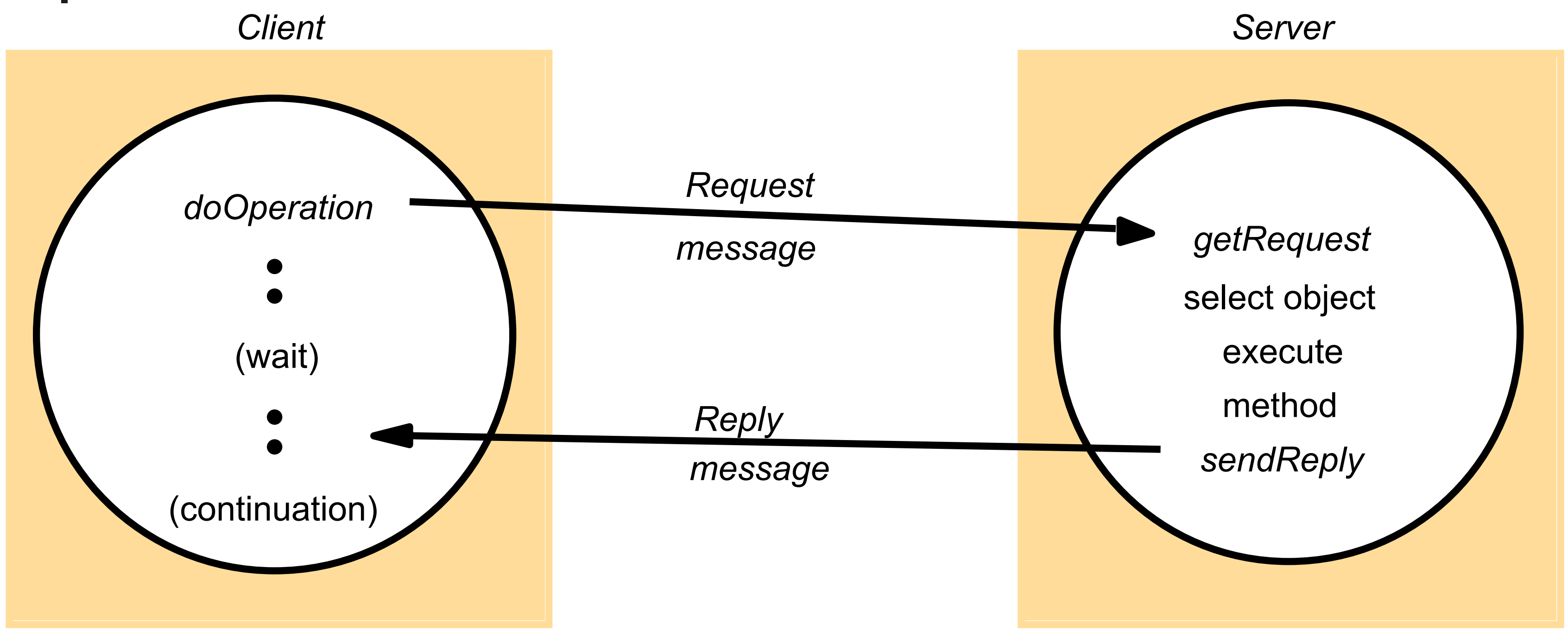


How does the layered design connect with A1-A4



- Wire protocol: A1 ad-hoc serialization, A2☐A4 Protocol buffers
- Dealing with faults:
  - A1, A2 – client side of request reply protocol.
  - A3, A4 – server side of request reply protocol
- Layer design:
  - What does each layer put on the wire?
  - Modularity / Abstraction
    - (vs. Cross-layer optimizations)

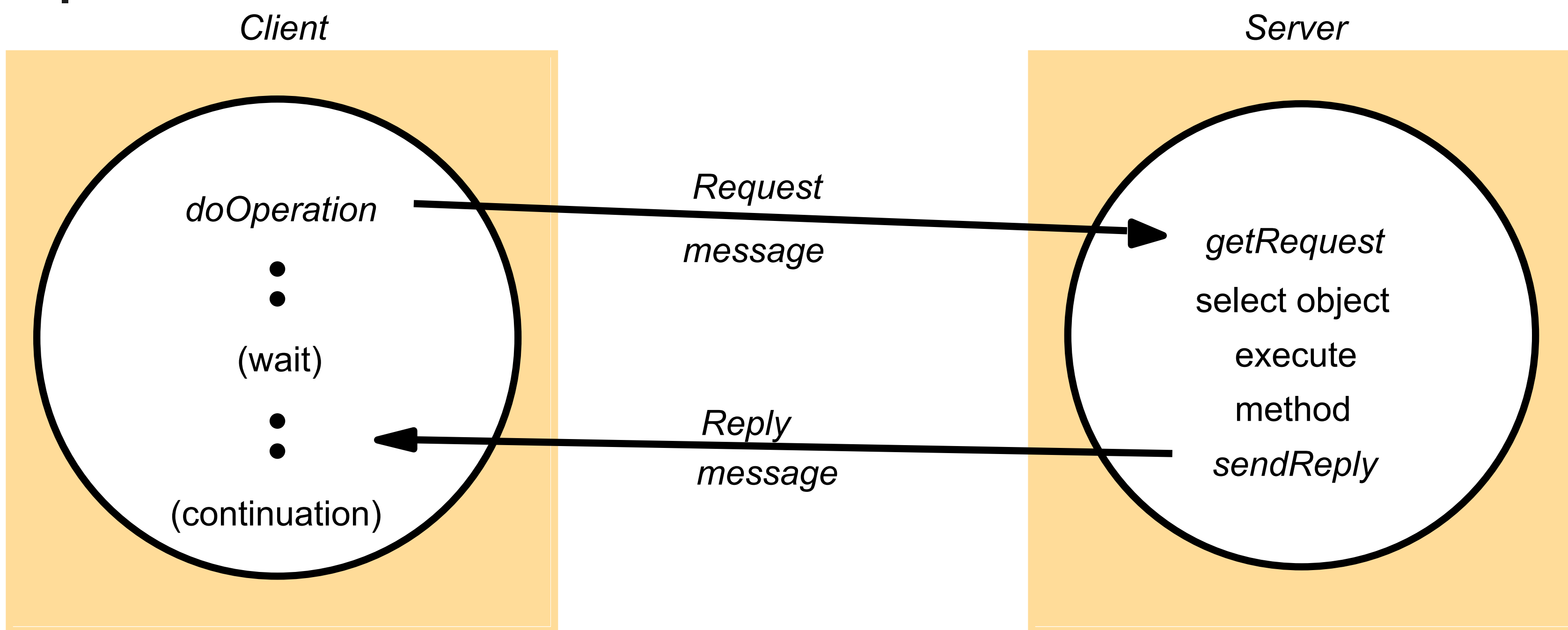
Request-reply Protocol



To design a request/reply protocol need to decide:

- transport protocol: TCP/UDP
- data format for serialization,
- exception handling,
- naming, …

Request-reply Protocol



Decide on:

- transport protocol: TCP/UDP?
- data format for serialization: Why? What are the alternatives? …
- fault handling,
- naming, …

Serialization: Protocol Buffers

```
message SearchRequest {
    string query = 1;
    int32 page_number = 2;
    int32 result_per_page = 3;
}
```

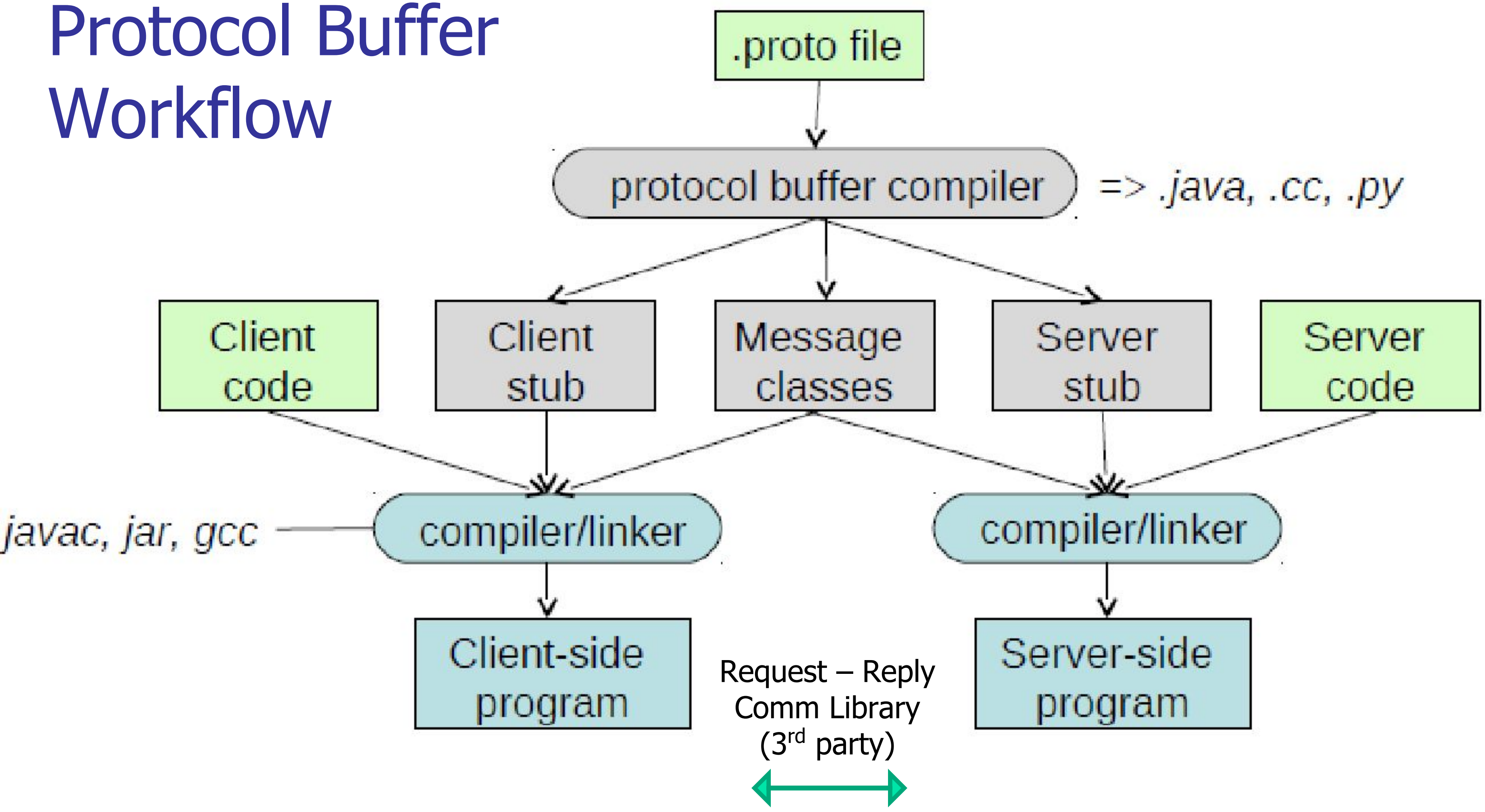
Properties:

- Efficient, binary serialization
- Support protocol evolution
  - Can add new parameters
  - Order in which parameters are specified is not important
  - Can skip non-essential parameters
- Supports types, which give you compile-time errors!
- Supports somewhat complex structures (embedded definitions)

Use:

- Pattern: for each new "service" define a message type for its input (the request) and one for its output (the reply) in a .proto file
- Used for other things, e.g., serializing data to non-relational databases –backward-compatible features make for nice long-term storage formats
- Google uses them *everywhere* (10,000s of proto buf definitions)

Protocol Buffer Workflow



Note: Serialization/deserialization only!

- Supports service definitions and stub generation …
- … but does not come with transport (your own code or libraries for this).

Protobuf vs. Alternatives

Q: What is the trade-off space?

- Protobufs are marshaled extremely effici
  - Binary format (as opposed to XML's textu
- Example (according to protobuf documentation):

- Runtime overheads
  - parsing speed,
  - space
- Ease of use
  - code maintainability
  - debugging cost

XML

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

- size: 69 bytes (w/o whitespaces)
- parse: 5,000-10,000ns

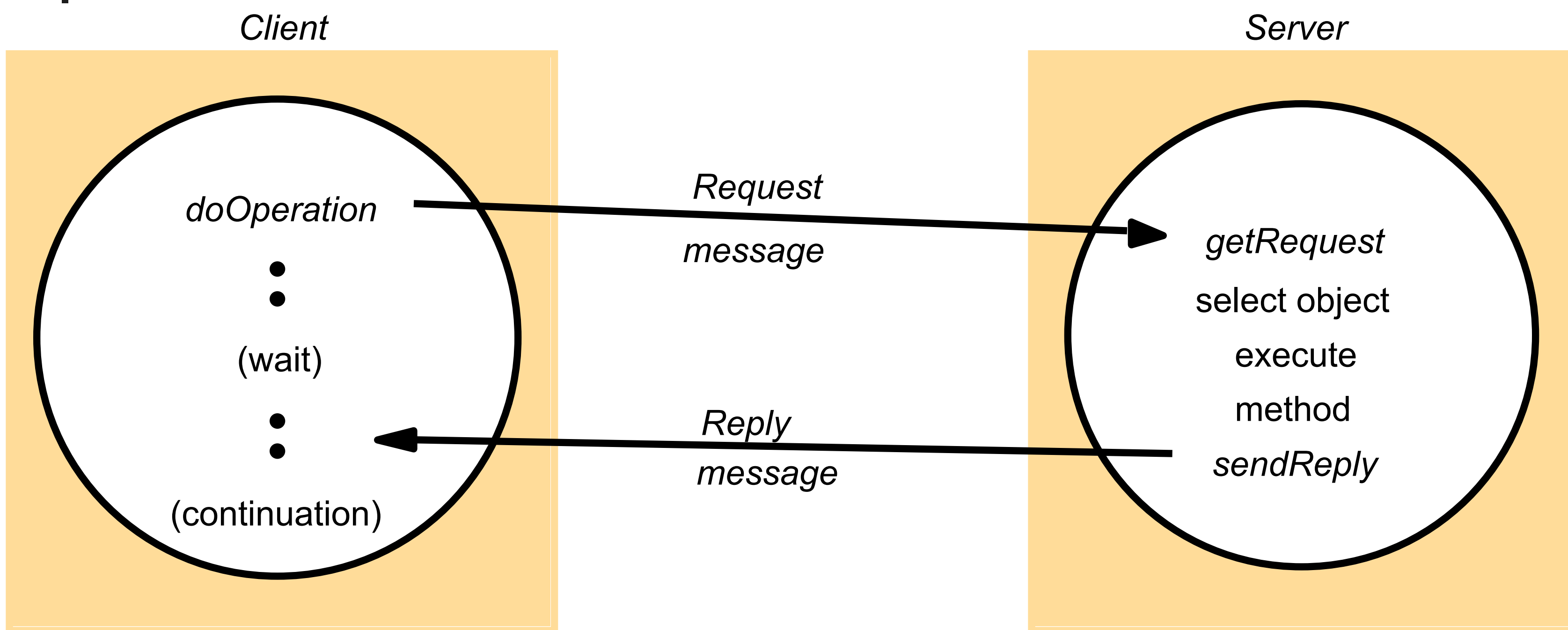
Protobuf

```
person {
    1:"John Doe"
    3:"jdoe@example.com"
}
```

- size: 28 bytes
- parse: 100-200ns

Any drawbacks?

Request-reply protocol



Design of a request/reply protocol: many decisions:

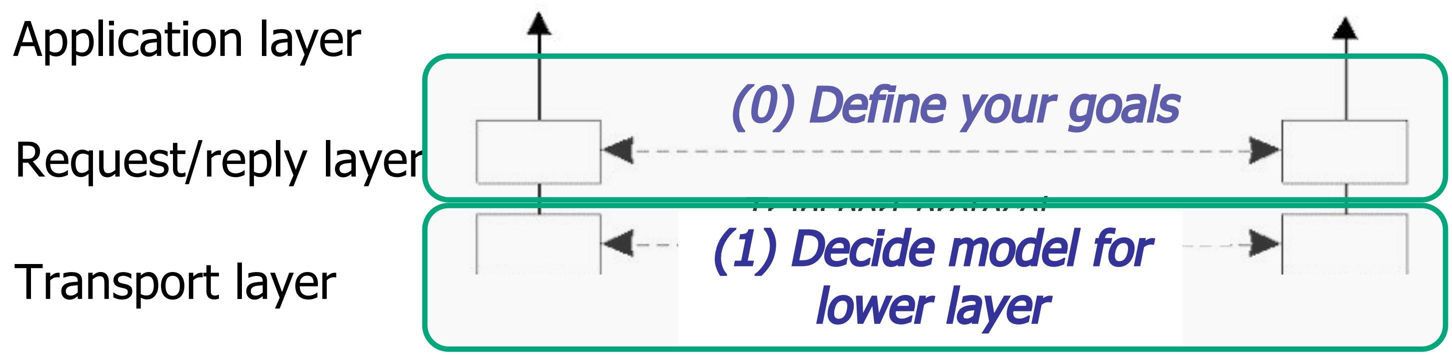
- transport protocol: TCP/UDP?
- data format for serialization: Why? Alternatives …
- fault handling,
- naming, …

Key Issue: Dealing with faults

Faults: crash, omission, timing, arbitrary

– some hidden by underlying network layers (e.g., TCP)

Design process in a layered system:



(0) ***Define your goals:*** Decide the properties your layer needs to offer

(1) ***Decide abstract model for environment*** (e.g., where does lower layer fail?)

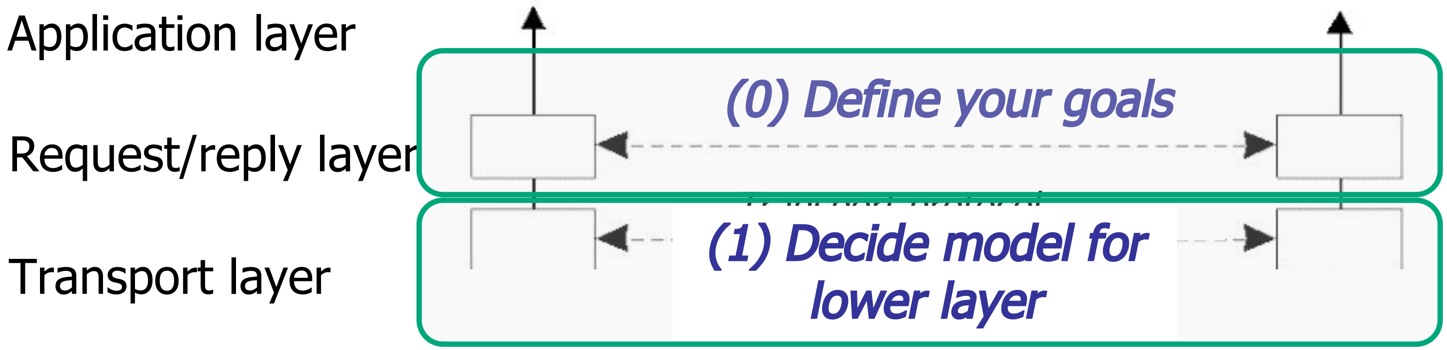
- E.g., communication is lossy, message propagation time is bounded or not, end-point failure mode (e.g., fail-crash, fail-silent)

(2) ***Design & implement your layer*** under the assumed model for underlying layer

(3) ***Deploy and test***

- <u>possibly iterate to (1)</u>: if it turns out assumptions about lower layer were incorrect
- you may need to redefine your goals and iterate back to (0) if it turns out implementation too complex or impossible with the assumptions made

Next slides will take you through this process …

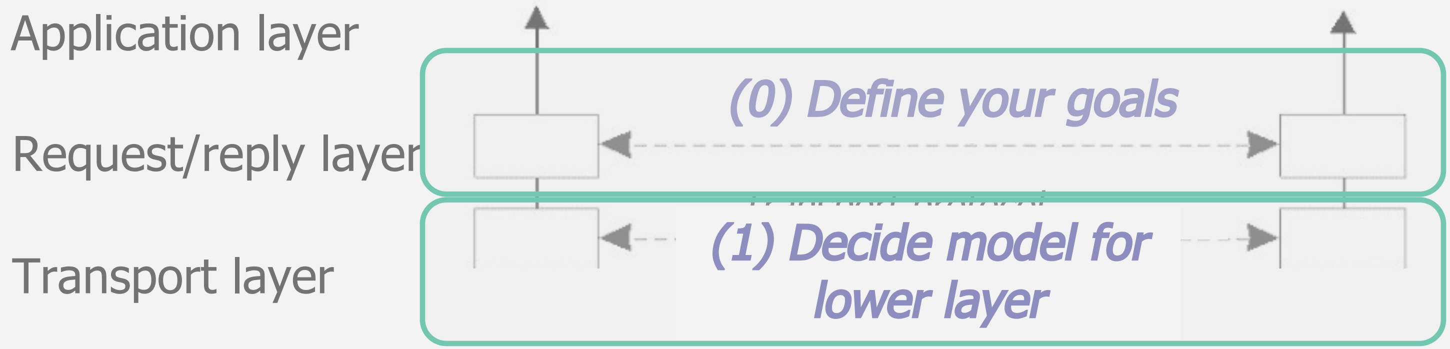


- Goal: Deliver "reliable" request-reply functionality
  - on top of some transport layer that passes messages
  - application agnostic (i.e., multiple applications can use it, no assumptions about the application)

- For each fault scenario

  (0): Further refine the goal: what exactly does "reliable" mean?

  (1): Possibly rethink assumptions about the transport layer/environment

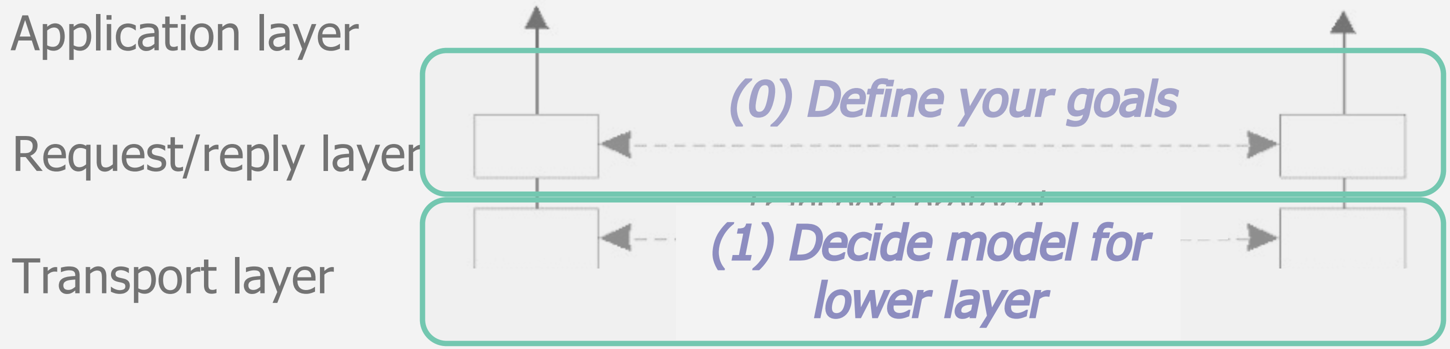


- Goal: Deliver "reliable" request-reply functionality
  - on top of some transport layer
  - application agnostic

Issue I – messages may be corrupted (Variant A)

- (0) Def. "reliable" = when received at destination, message is unaltered
- (1) Lower layer model: No message loss. Bounded propagation time. No end-point failure. Random bit flips (with some low probability)

Q: Design a protocol that provides the above definition of 'reliable' request/reply communication

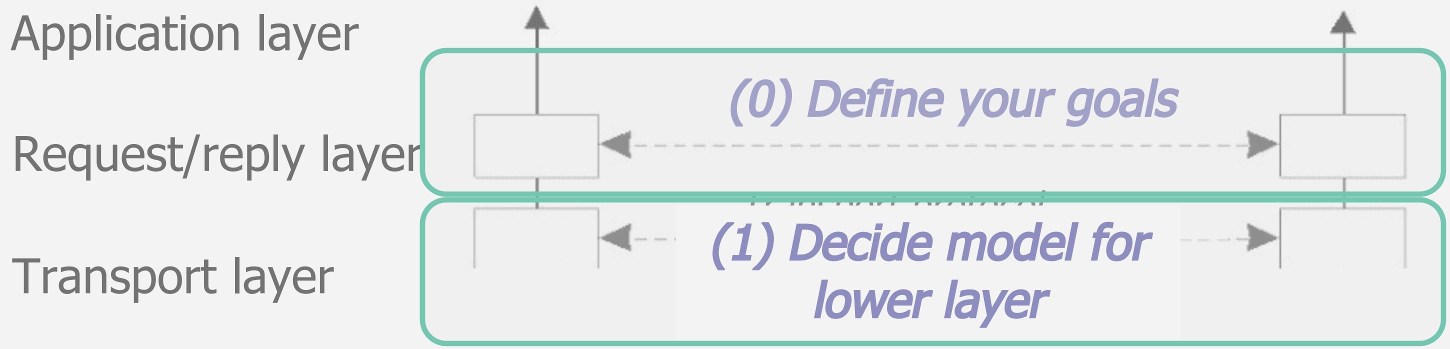


- Goal: Deliver "reliable" request-reply functionality
  - on top of some transport layer
  - application agnostic

Issue I – messages may be corrupted (Variant **B**)

- (0) Def. "reliable" = when received at destination, message is unaltered
- (1) Lower layer model: No message loss. Bounded propagation time. No end-point failure. Adversary can rewrite packets.

Q: Design a protocol that provides the above definition of 'reliable' request/reply communication



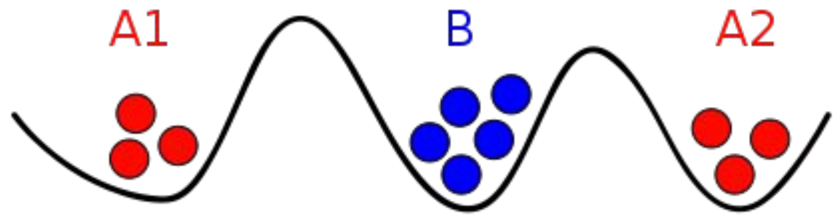
- Goal: Deliver "reliable" request-reply functionality
  - on top of some transport layer
  - application agnostic

Issue II: Message loss

- (0) "reliable" = the request is ***always*** delivered to the destination and a reply is ***always*** received by sender. Bounded termination time.
- (1) Lower layer model: Transport layer has message loss. Message propagation time is bounded. No end-point failures.

Q: Design a protocol that provides the above definition of 'reliable' request/reply communication

Two Generals Problem



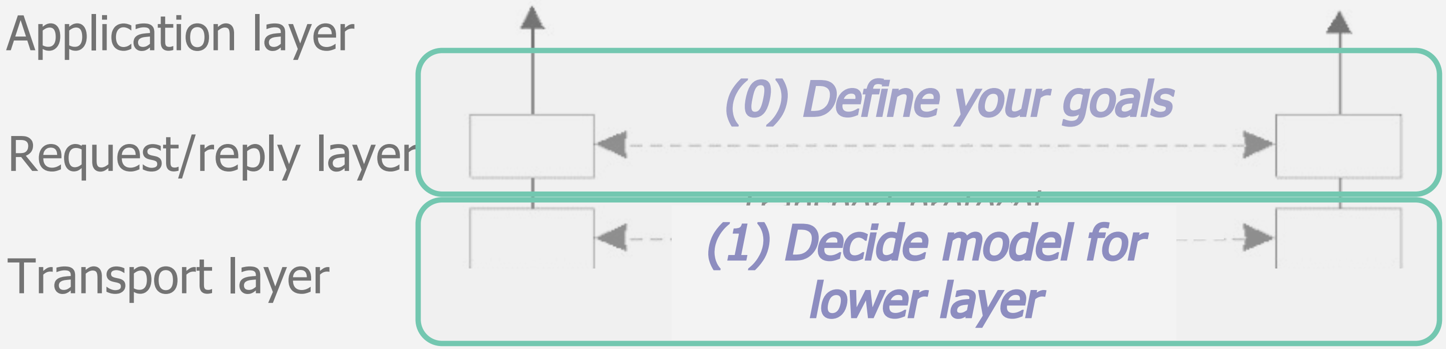
Context

- Red generals need to attack at the same time (coordinate) to win the battle.
- Coordination is possible only by sending a messenger through the territory controlled by the Blue general (who might capture the messenger)

Task: Design a strategy that guarantees that the Red generals can <u>always</u> coordinate their attack

Takeaway:

No solution that guarantees *correctness* <u>**AND**</u> *termination*

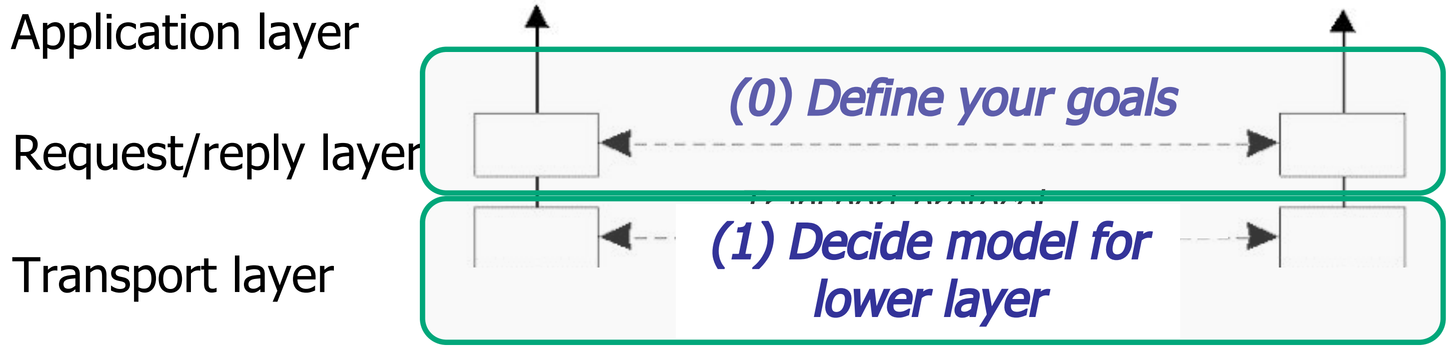


- Goal: Deliver “reliable” request-reply functionality
  - on top of some transport layer
  - application agnostic

Issue II: message loss, propagation time,

- (0) “reliable” = the request is *always* delivered to the destination and a reply is *always* received by sender. Bounded termination time.
- (1a) Lower layer model: Transport layer has loss. Message propagation time is bounded. No end-point failures.
- (1b) Lower layer model: no message loss, but UNBOUNDED message propagation time and end-point-failures.

IMPOSSIBLE



- Goal: Deliver "reliable" request-reply functionality
  - "reliable" = the request is ***always*** delivered to the destination and a reply is ***always*** received by sender. Bounded termination time

IMPOSSIBLE

Not practical
(Duplicates / Termination)

- Redefine meaning of "reliable" / new semantics
  - at-least-once: *if* protocol completes, the message has been delivered to the destination (at least once). [no bound for termination]
    - possible variant: exactly-once
  - at-most-once: ***when*** the protocol completes (i.e., fixed # of retries), the message has been delivered to the destination at most once.
    - If 'success': the message has been delivered to the destination (once).
    - If 'failure': the message may have been delivered to the destination (once)

Widely used

Let’s recap where we are …

Goal: “Reliable” request-reply functionality.

[deals ‘correctly’ with message corruption, duplication, loss, server crashes]

BUT … one can not build a protocol that <u>guarantees</u> both termination AND “reliable” message delivery on top of an unreliable substrate.

- As a consequence
  - Redefined expectations: ‘reliable’ ☐ at-most-once

Next:

- Impact of at-most-once semantics
- Implementation

W2-Q0: Request/reply semantics

Can my [correctly implemented] request-reply functionality at-most-once semantics still lead to surprises for application developers?

Assume I have implemented a key/value store on top of it.

```
PUT (key, 100)
PUT (key, 1)
val = GET (key)
```

First two PUT operations **succeed**.

What value is associated with `key` (no value associated initially)

Choose: nothing, 0, 1, 100, 99, something else?

W2-Q1: Request/reply semantics”

Can my [correctly implemented] request-reply functionality at-most-once semantics still lead to surprises for application developers?

Assume I have implemented a key/value store on top of it.

```
PUT (key, 100)
PUT (key, 1)
val = GET (key)
```

First two PUT operations **fail**.

What value is associated to `key` (no value associated initially)

Choose: nothing, 0, 1, 100, 99, something else?

W2-Q2: A counter on top of at-most-once

Assume a counter serving *increment* and *decrement* requests correctly implemented over a request-reply protocol with ***at-most-once*** semantics

The protocol used to send 10 requests to increment a counter (initially at 0). All return **FAIL**.

Select all possible values of the counter at the end of this sequence.

Select from: 0 / 1 / 10 / 20

W2-Q3: A counter on top of at-least-once

Assume a counter serving *increment* and *decrement* requests correctly implemented over a request-reply protocol with ***at-least-once*** semantics

The protocol used to send 10 requests to increment a counter (initially at 0). All return **SUCCESS**.

Select all possible values of the counter at the end of this sequence.

Select from: 0 / 1 / 10 / 20

W2-Q4: Testing

The TA wants to test correctness of your key/value sever developed on top of a request/reply protocol with at-most-once semantics.

The client API (put, get) returns four possible application-level error codes

- 0 – success, 1 – timeout , 2 – key is not there, 3 – server error

You are graded based on the following test:

```
errCode1 = put (K,  V)
(errCode2, VReturned) = get (K)
if (errCode1 == SUCCESS and
  errCode2 == SUCCESS and
  V == VReturned)
then TEST_PASSED  add points
else TEST_FAIL   deduct points
```

Q: Is this correct? Should you complain?

What should be the decision after previous test?

PUT returns / GET returns	SUCCESS	TIMEOUT	NO-KEY	FAIL CODE (explicit info that the server has rejected the request)
SUCCESS	PASS (if V == Vreturned)	PASS (if V == Vreturned)	FAIL	
TIMEOUT	UNDECIDED	UNDECIDED		
NO-KEY	FAIL	UNDECIDED		
FAIL	FAIL	FAIL		

Let’s recap where we are …

Goal: “Reliable” request-reply functionality.

[deals ‘correctly’ with message corruption, duplication, loss, server crashes]

BUT … one can not build a protocol that guarantees termination AND reliable message delivery on top of an unreliable substrate.

- As a consequence
  - Redefined expectations: ‘reliable’ ☐ at-most-once

Next:

- Impact of at-most-once semantics
- Implementation issues

- Goal: Deliver at-most-once request-reply functionality
  - on top of some transport layer, application agnostic

Implementation issues:

- [client side] Fairly simple: retry policy, pairing requests and replies, identify message corruption
- [server side] Tricky. Main issue: Caching / filtering

  [Why do you need caching/filtering?]
  - Issue 1: What to cache? [i.e., What 'state' to maintain to enable filtering?]
  - Issue 2: How does one garbage collect the cache? (Eviction Policy)
    - Client and server need to agree on retry policy (#retries / timeout)
    - Additional assumptions about the underlying transport layer: maximum message propagation time in the network

W3-Q1: Cache sizing

A key-value server is migrated on a faster node. Tests with a wide set of benchmarks show that code runs at least 2x faster

On the original node the server was configured with a 500MB cap on the space to maintain state to deal with duplicate requests (the "cache"). Everything was working fine.

On the new node the same configuration is maintained.

Q: Will the k/v store throughput on the new node double?

- Yes, of course – why not?
- Nah, this is too optimistic – things never work the way one hopes
- Maybe, it depends on …?

- Goal: Deliver at-most-once request-reply functionality
  - on top of some transport layer, application agnostic

Implementation issues:

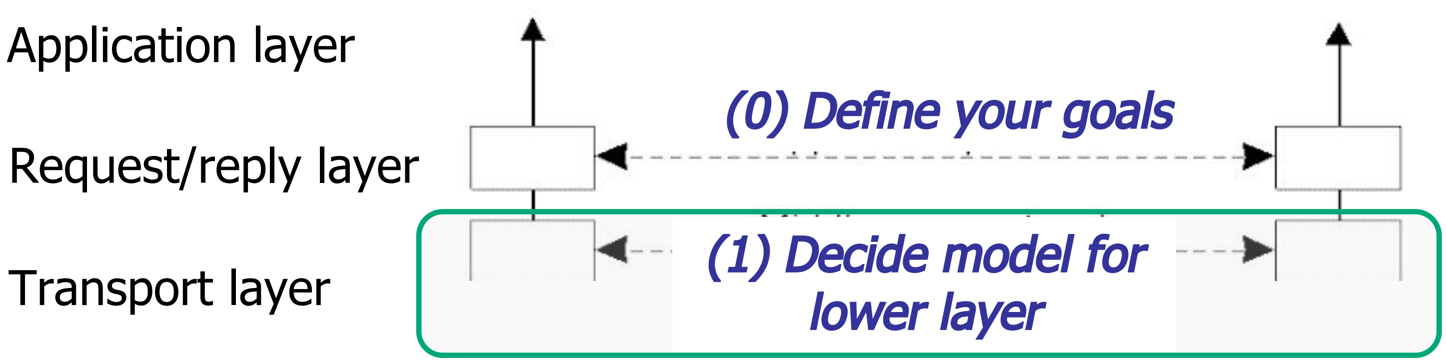
- [client side] Fairly simple: retry policy, pairing requests and replies, identify message corruption
- [server side] Tricky. Main issue: Caching / filtering

  [Why do you need caching/filtering?]
  - Issue 1: What to cache? [i.e., What 'state' to maintain to enable filtering?]
  - Issue 2: How does one garbage collect the cache? (Eviction Policy)
    - Client and server need to agree on retry policy (#retries / timeout)
    - Additional assumptions about the underlying transport layer: maximum message propagation time in the network
  - Issue 3: Cache sizing
    - Impact on throughput!

Emergent property

Recap main ideas (I)

- Design process for layered systems:
  - (0) Decide the properties top layer needs to offer
  - (1) Decide abstract model for environment (lower layer)
    - E.g., communication is lossy, message propagation time is unbounded, end-point failure mode (fail-crash, fail-silent)
  - (2) Design solution under the assumed model
  - (3) Deploy and test
    - possibly iterate to (1) or even redefine your goals and iterate back to (0)



Recap main ideas (II)

Observation: One can not build a protocol that <u>guarantees</u> both termination AND “reliable” message delivery on top of an unreliable substrate.

Solution in the context of request-reply functionality:

- (re)define semantics or “reliable”: at-most-once

- Key ingredients for implementation:
  - unique messageIDs, checksums,
  - Server side mechanism to deal with client retries (“caching”): (i) messageID to filter out duplicates; (ii) cache reply value to reply to identified duplicate requests that have already been served.
  - client and server <u>must</u> coordinate: #retries / retry timeout
  - assumption about maximum message propagation time,

Recap main ideas (III)

- Design guidelines:
  - (1) where possible design systems based on timeouts rather than explicit message exchange
    - CAN NOT guarantee both termination and reliable message delivery anyways
  - (2) simpler designs are possible if one can make assumptions about the application (layer above)

A few additional design issues

- The design can be simplified if the lower layer can make assumptions about the usage patterns
- Breaking the abstraction: Layering vs. cross-layer optimizations
- Preserving at-most-once semantics across server crashes

A few additional design issues

- The design can be simplified if the lower layer can make assumptions about the usage patterns
- Layering vs. cross-layer optimizations
- Preserving at-most-once semantics across server crashes

There are situations where the design may be simpler

- Goal so far: Deliver "at-most-once" request-reply functionality
  - on top of some transport layer
  - ~~application agnostic~~

Can the design be simplified …

- … if server-side is stateless?
  - i.e., no state kept at the server across requests
    - E.g., here is an integer; give me the next prime number.
  - implication: does not care about duplicate requests, server crashes

Warning: Needs careful analysis!

- Goal so far: Deliver "at-most-once" request-reply functionality
  - on top of some transport layer
  - ~~application agnostic~~

Can the design be simplified …

- … if requests are idempotent?
  - Idempotent: the result of a successfully performed request is independent of the number of times it is executed. (i.e., same result returned and the same end state).
    - Sequence: PUT (key, 1); val2=GET(key); val1=GET(key);

W3-Q2: Idempotent requests

Based on the fact that a key-value store PUT/GET always return the same result (no mater how many retries) you try a simpler implementation for a request/reply layer.

- up to 3 retries
- at the server: no attempt to identify retries (just re-execution).

Assume a key/value store on top of this request/reply layer

(initially nothing is associated with the 'key' and the following trace)

```
PUT (key, 100)
PUT (key, 1)
val = GET (key)
```

Q: All operations ***succeed***. What are the possible values in `val`?

1 (one)   100   something else   nothing

Poll: Idempotent requests

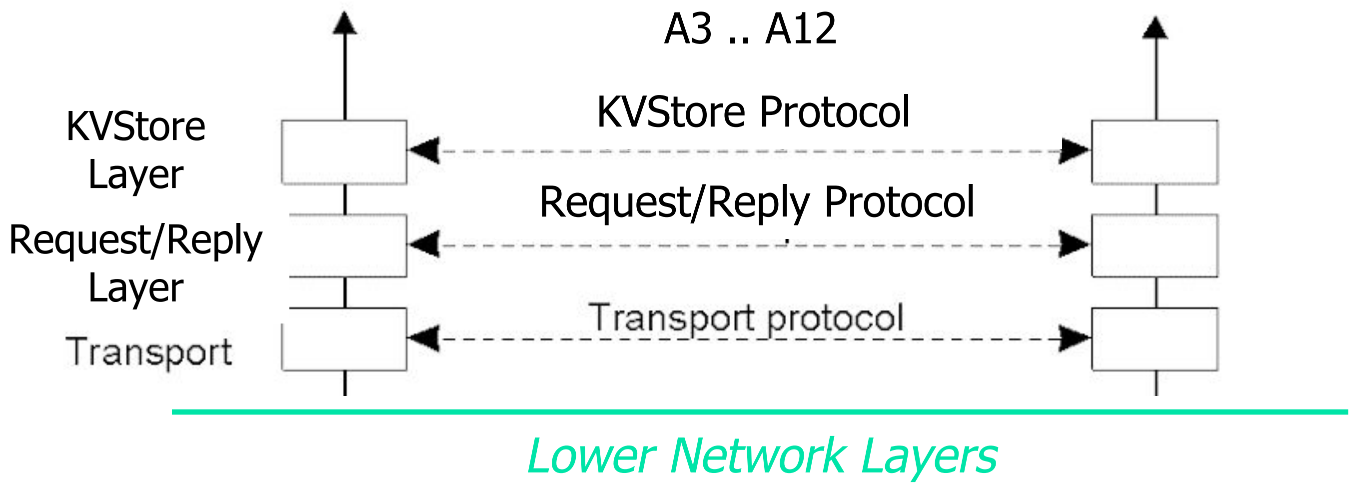
You are asked to implement a request-reply protocol specialized for applications that will only have idempotent requests.

Q: Which one of the following is true?

- [] You can safely get rid of the component that identifies duplicate requests. There will be no surprises at the application level.
- [] Still need to filter out duplicates. (explain why)

A few additional design issues

- Generally the design can be simplified if the lower layer can make assumptions about the usage patterns
- Layering vs. cross-layer optimizations
- Preserving at-most-once semantics across server crashes



Q: Can you come up with a cross-layer optimization that provides some tangible benefit (e.g., performance, efficiency) for your KVS server but still maintain at-most-once semantics?

Poll

Your key value store may receive up to 4 GET (re)tries with the same messageID (and, implicitly the same key) within the cache timeout period.

Which implementation would you go for?

(Select one of the following)

- Cache the value at the time of the first GET (re)try, and return that same value for all other retries.
- Always return the latest value in the KV store at the time of the retry.

A few additional design issues

- The design can be simplified if the lower layer can make assumptions about the usage patterns
- Layering vs. cross-layer optimizations
- Preserving at-most-once semantics across server crashes

One more server side issue: # How to handle server crashes?

Why is this a problem?

- Solution 1: Persist all cache transactions. But slooooow!
- Solution 2: Once server restarts, delay server does not reply to requests for a period of time T.
  - What's the minimal **T** that guarantees correctness?
  - **W3-Q3**
- Solution 3: Can you design an even better solution?
  - Goal: Based on Solution 2, preserve at-most-once semantics and improve server availability (lower wait time to restart), minimal overhead

W2-Q7: At-most-twice semantics 😊

Consider the following trace. All requests calls are synchronous. There is nothing associated with K in the beginning.

PUT (K, 10)
PUT (K, 20)
val = GET (K)

The underlying request/reply protocol over which the key value store was implemented offers a novel *"at-most-twice"* semantics (claimed to be easier to implement):

Let's be more formal and define at-most-twice semantics as follows: 0, 1 or 2 deliveries may all correct depending on the message loss scenario.

* The request-reply layer guarantees that in all scenarios at most two copies of the request are delivered to the server.

* The client returns TIMEOUT if it can not guarantee that at least one copy of the request has been received (similar to at-most once).

* The client returns SUCCESS if it can guarantee at least one copy of the request has been received.

In the code snippet above all commands return **SUCCESS.**

What are the possible values in *val? (Select all that apply)*

□ empty □ 10 □ 20 □ 99

- Now-think about "at-least-once" implications
- Or exactly-once: How big is the cache then?

Backup slides: Dealing with failures – for request/reply

Failures: crash, omission, timing, arbitrary

– some hidden by underlying network layers (e.g., TCP)

What can go wrong:

1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

Backup slides: Dealing with failures (1/5)

What can go wrong with RPC:

1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

[1:] Client cannot locate server. Relatively simple ☐ just report back to client application

Backup slides: Dealing with failures: message loss (2/5)

[2, 4:] Client request lost or server reply is lost

Two cases:

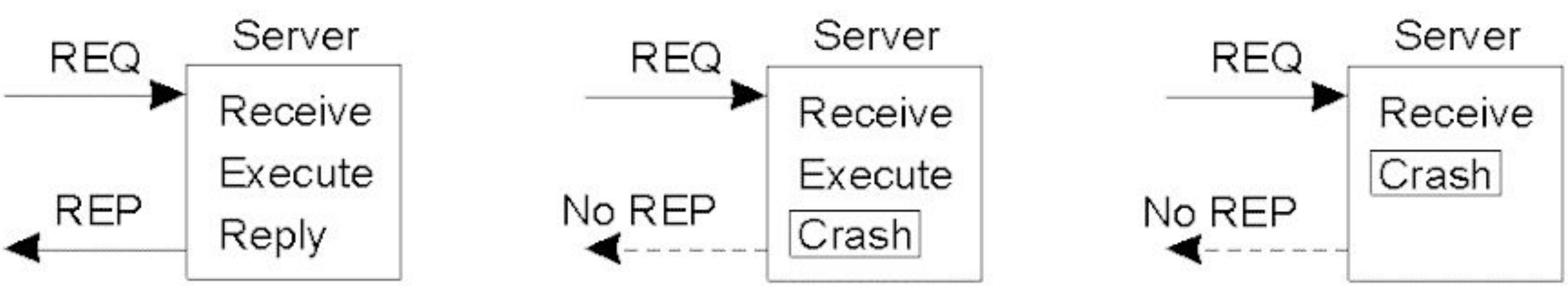
- Idempotent requests (i.e., server state does not change)
  - Just resend. But: how many times?
  - Can this lead to reordering visible by the client?
- Non-idempotent requests,
  - Add request identifiers so that you can repeat invocation
  - Leads to more complex mechanism:
    - What should the server log? When?
    - Now one has to deal with state at the server (the log).
      - New issue: how to maintain this state? Log can not grow infinitely

Midterm-like question: sketch a mechanism to deal with lost messages

Backup slides:
Dealing with failures: machine crashes

[3] Server crashes

- Same issues as for message loss: client does not know what the server has already done:
- Additional issue: state loss



No-failure execution

Client can not distinguish between these two

Midterm-like question: sketch a mechanism to deal with server crashes

Backup slides: Dealing with failures: node failures

[5:] Client crashes □ Issue: The server is doing work and holding resources for nothing (**orphan** computation).

Possible solutions:

- [expiration] Require computations to complete in a *T* time units. Old ones are simply removed.
- [orphan extermination] Orphan is killed by client when it reboots