Networking (2)

Acks

- When a segment is received, the highest permissible Ack is sent back
  - if data up through i has been received, the ack sequence number is i+1
  - if data up through i has been received, as well as i+100 through i+200, the ack sequence number is i+1
    - a higher value would imply that data in the range [i+1, i+99] has been received
- Every segment sent contains the most up-to-date Ack

Quiz 1

A TCP sender has sent four hundred-byte segments starting with sequence numbers 1000, 1100, 1200, and 1300, respectively. It receives from the other side three consecutive ACKs, all mentioning sequence number 1100. It may conclude that

a) The first segment was received, but nothing more

b) The first, third, and fourth segments were received, but not the second

c) The first segment was received, but not the second; nothing is known about the others

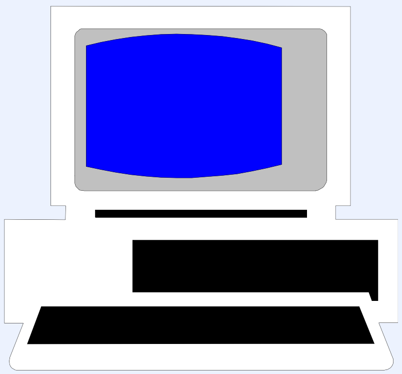
d) There's a bug in the receiver

Fast Retransmit and Recovery

- Waiting an entire RTO before retransmitting causes the “pipeline” to become empty
  - must slow-start to get going again
- If one receives three acks that all repeat the same sequence number:
  - some data is getting through
  - one segment is lost
  - immediately retransmit the lost segment
  - halve the congestion window (i.e., perform congestion control)
  - don’t slow-start (there is still data in the pipeline)

Remote Procedure Call Protocols

Local Procedure Calls

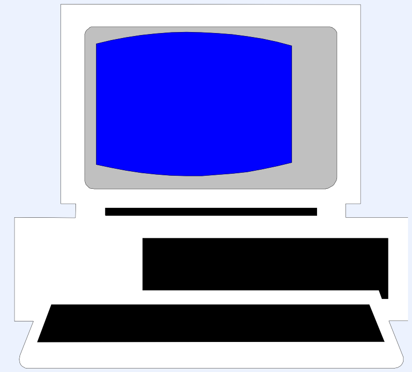


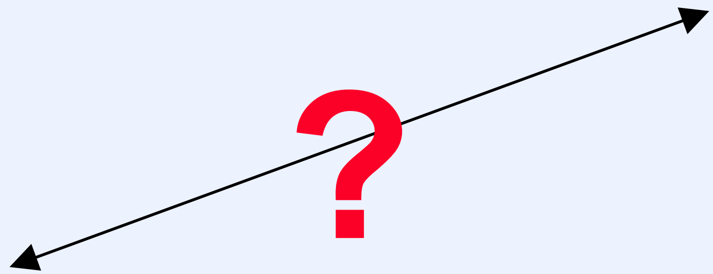
```
// Client code
     . . .
result = procedure(arg1, arg2);
     . . .
```

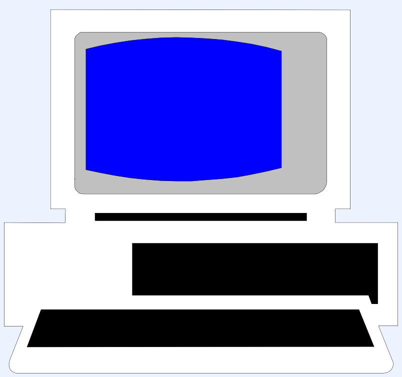
```
// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
     . . .
     return(result);
}
```

Remote Procedure Calls (1)

```
// Client code
    . . .
result = procedure(arg1, arg2);
    . . .
```





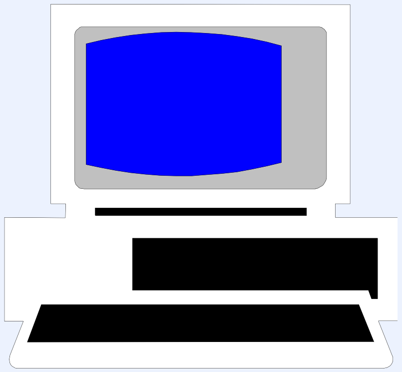


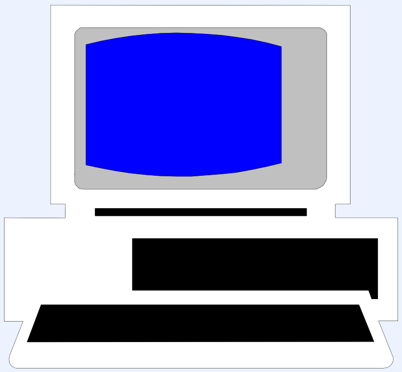
```
// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
    . . .
    return(result);
}
```

Remote Procedure Calls (2)

```
// Client code
    . . .
result = procedure(arg1, arg2);
    . . .
```

Client-Side Stub

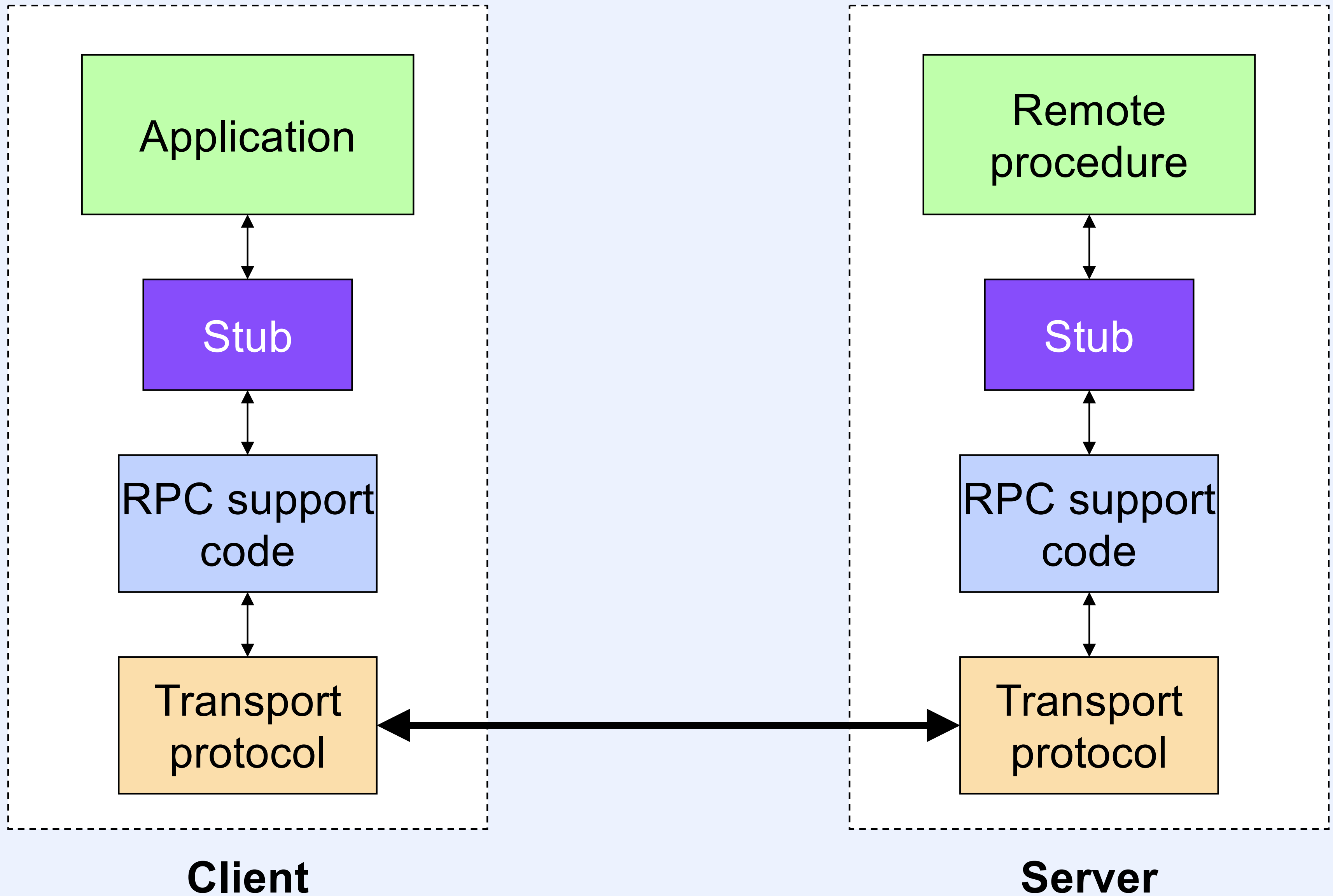




Server-Side Stub

```
// Server code
result_t procedure(a1_t arg1, a2_t arg2) {
    . . .
    return(result);
}
```

Block Diagram



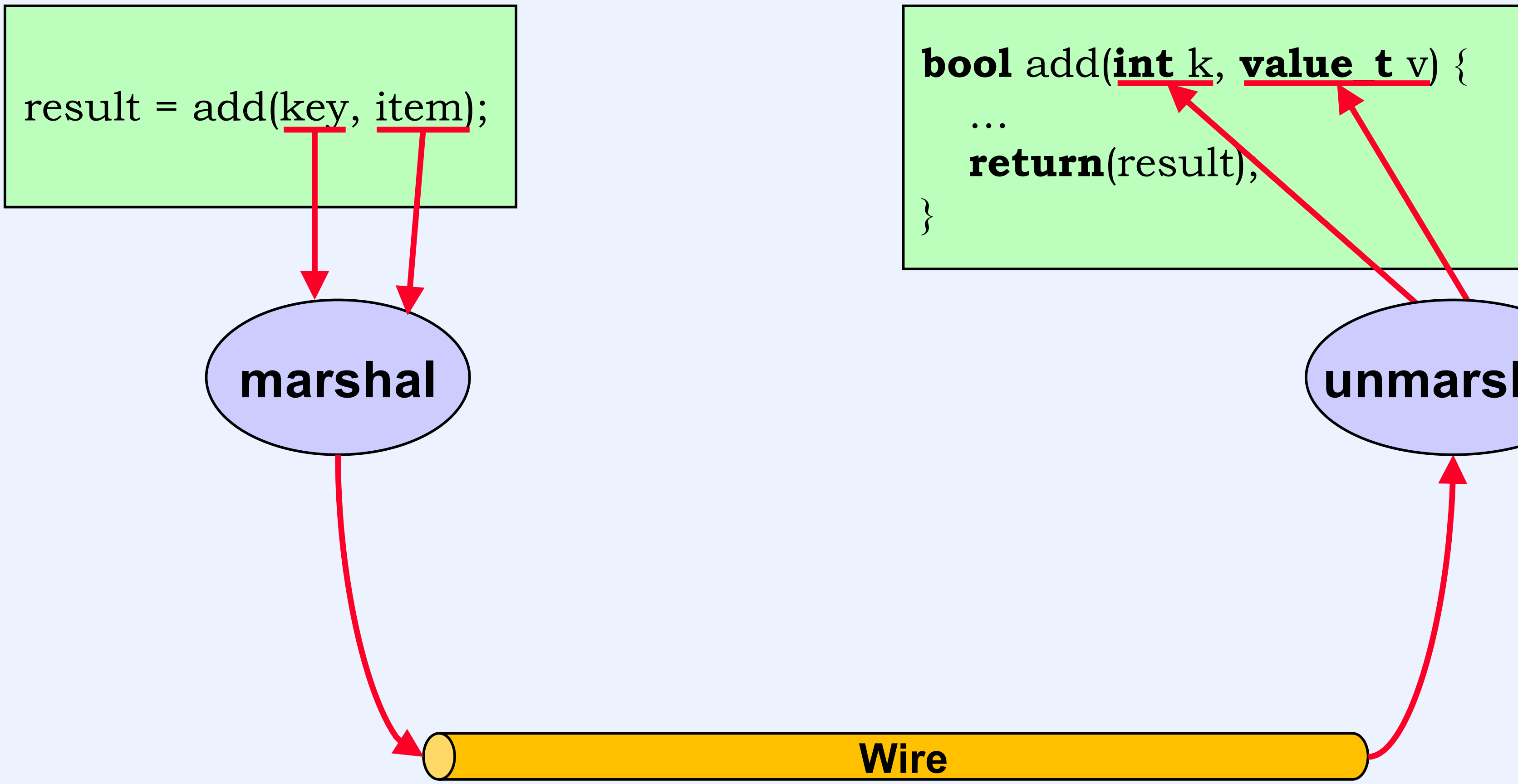
ONC RPC

- Used with NFS
- eXternal Data Representation (XDR)
  – specification for how data is transmitted
  – language for specifying interfaces

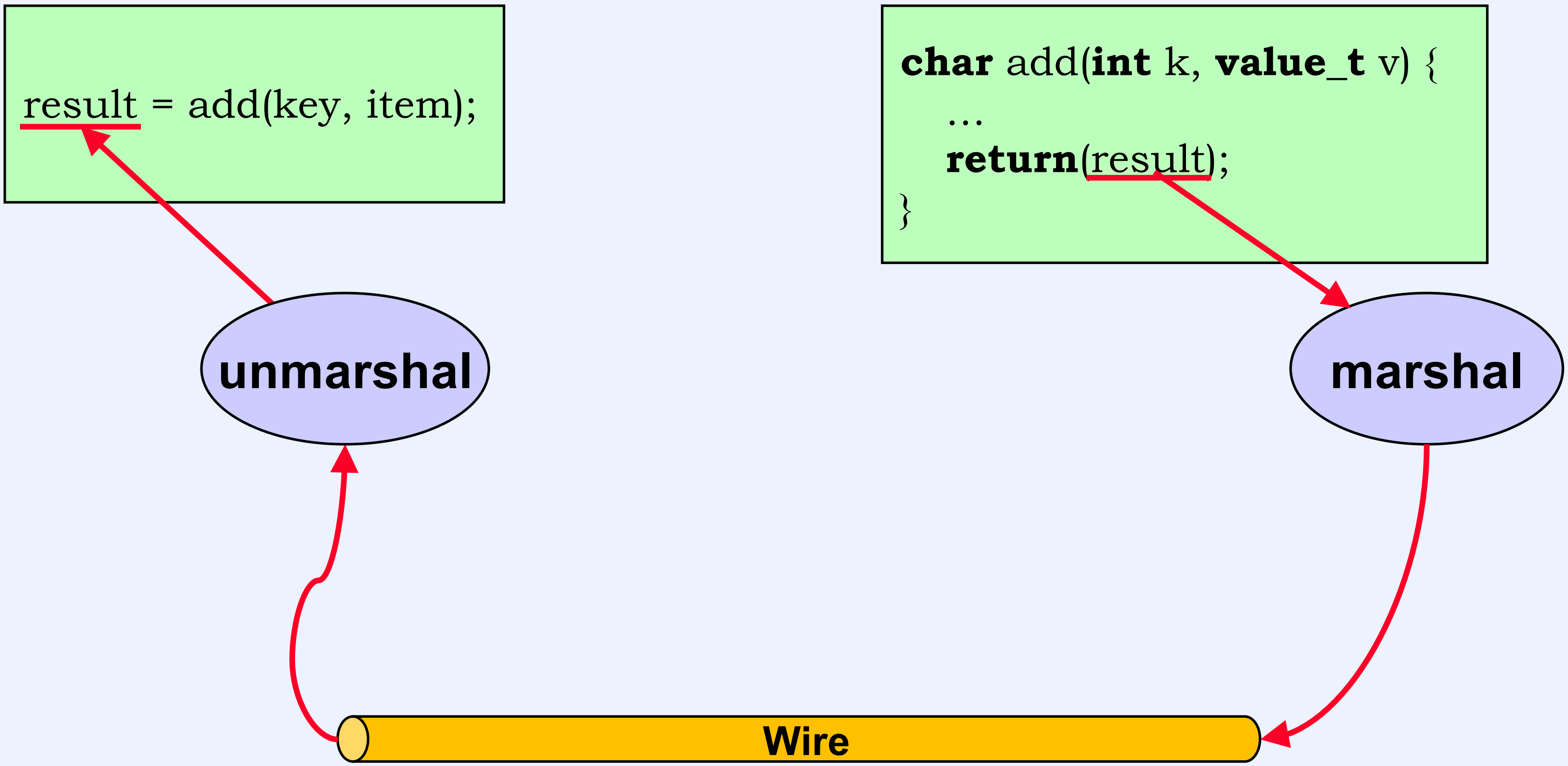
Example

```
typedef struct {
   int   comp1;
   float comp2[6];
   char  *annotation;
} value_t;

typedef struct {
   value_t  item;
   list_t   *next;
} list_t;

bool add(int key, value_t item);
bool remove(int key, value_t item);
list_t query(int key);
```

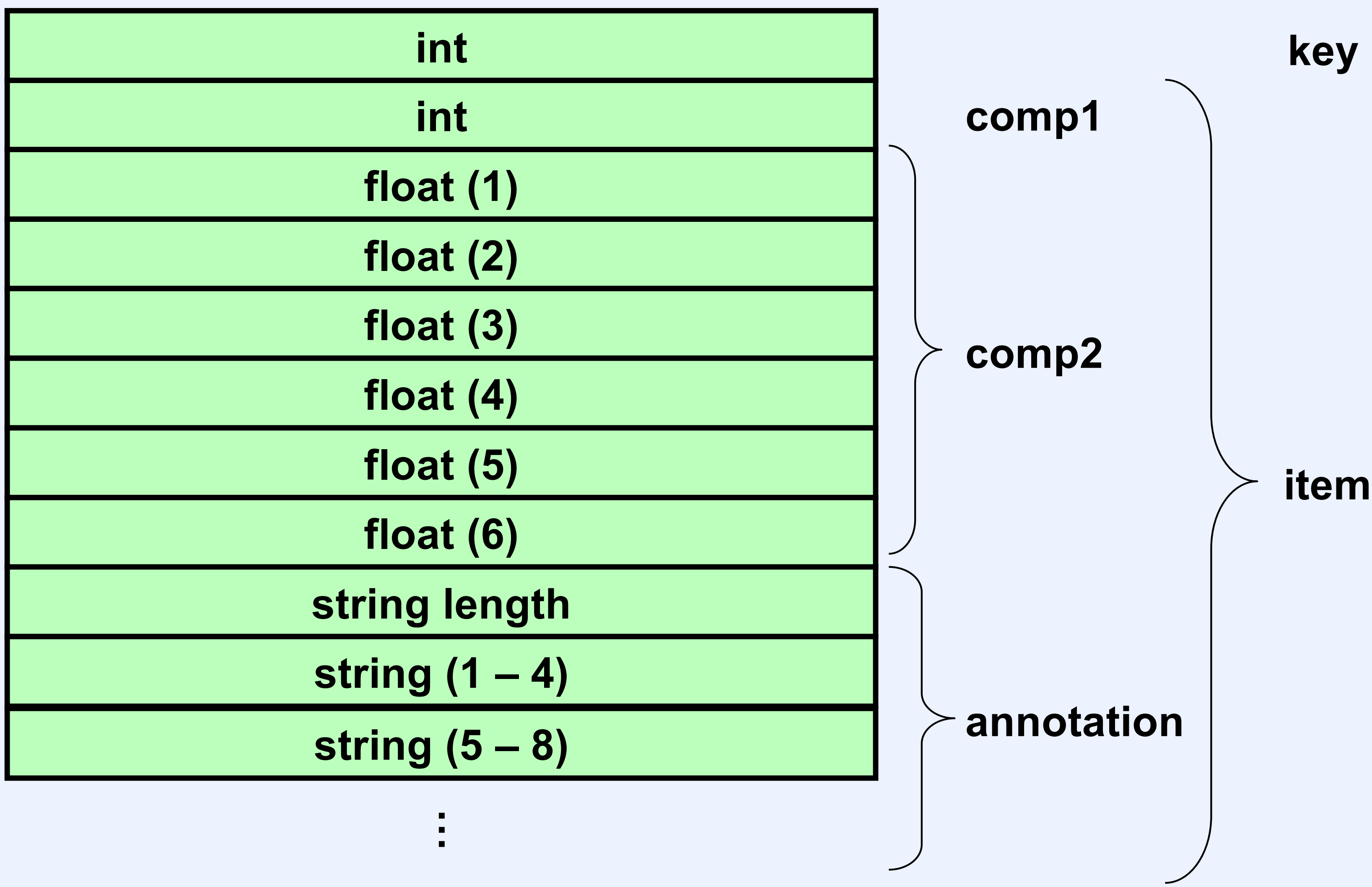
Placing a Call



Returning From the Call



Marshalled Arguments



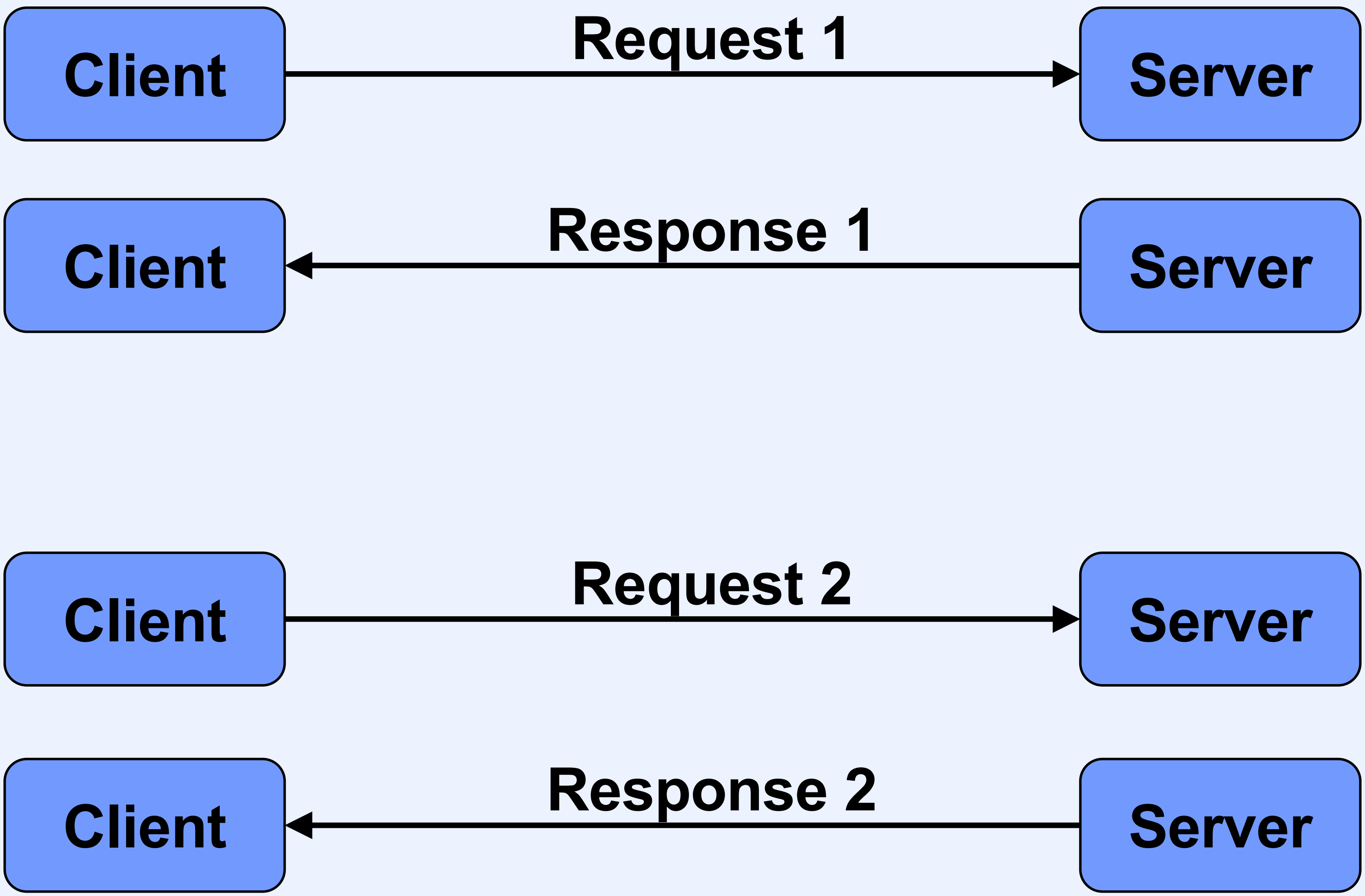
Marshalled Linked List

		array length
0:	value_t	next: 1
1:	value_t	next: 2
2:	value_t	next: 3
3:	value_t	next: 4
4:	value_t	next: 5
5:	value_t	next: 6
6:	value_t	next: -1

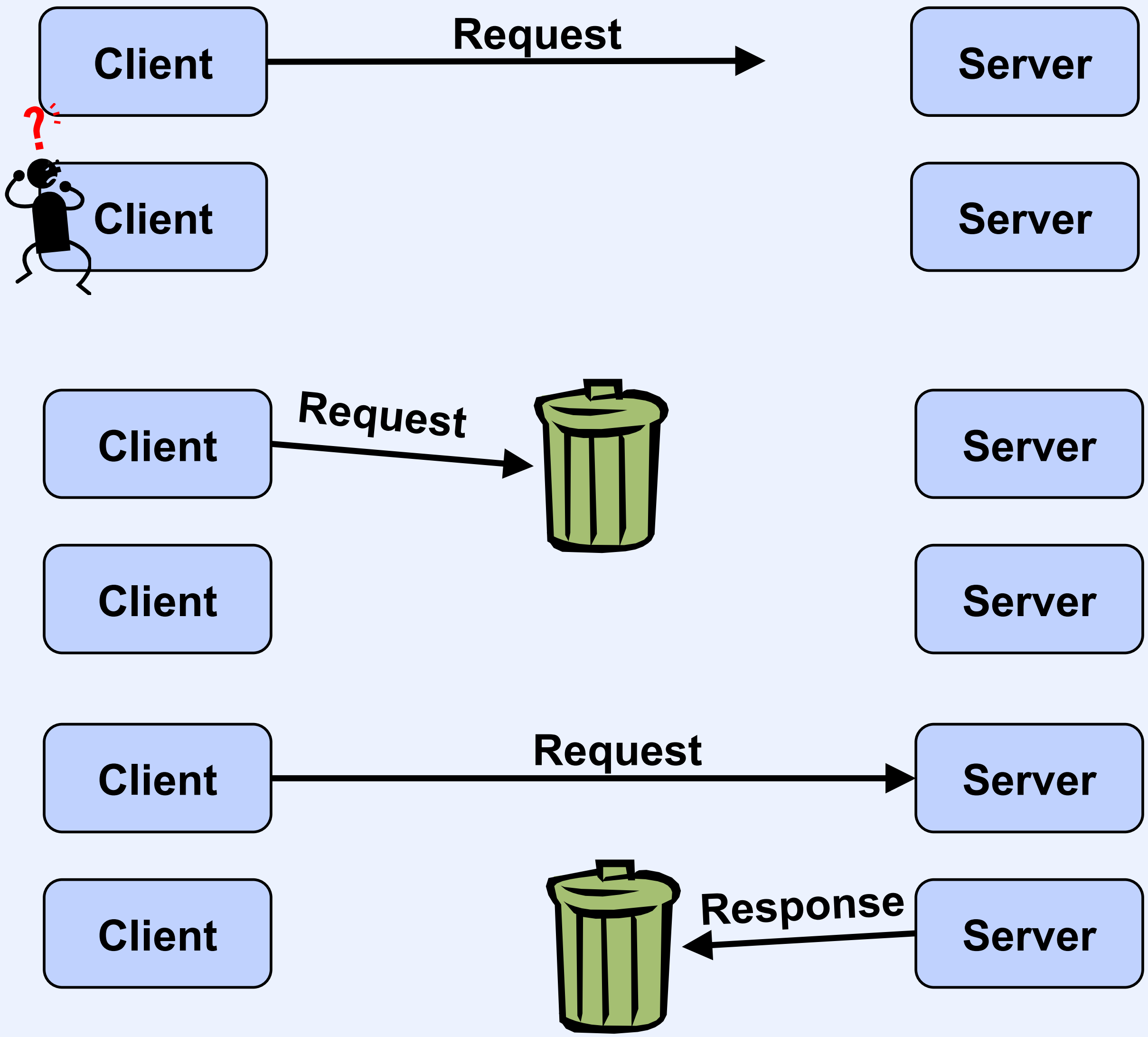
Reliability Explained …

- Assume, for now, that RPC is layered on top of (unreliable) UDP
- Exactly once semantics
  - each RPC request is handled exactly once on the server

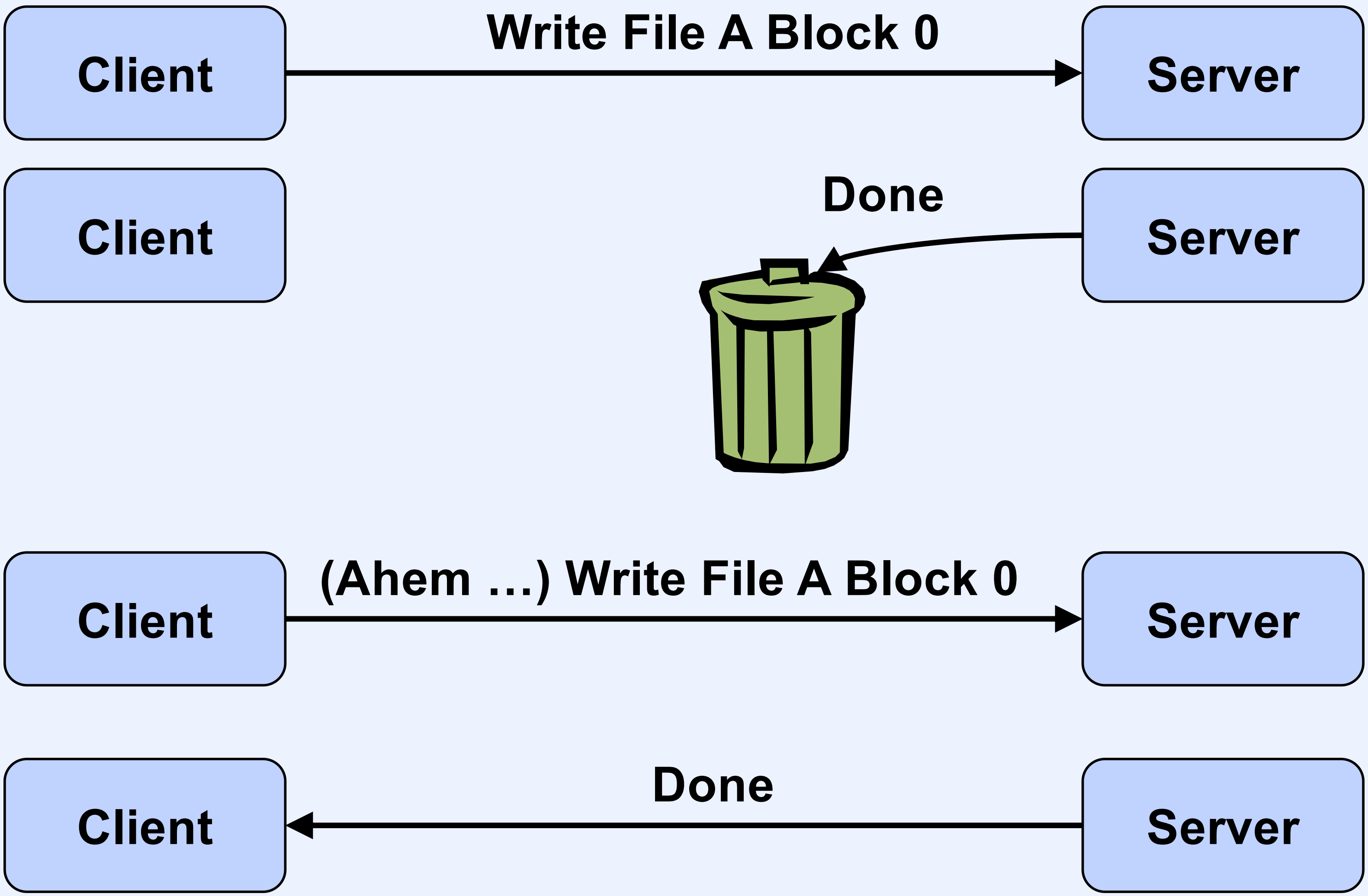
RPC Exchanges



Uncertainty

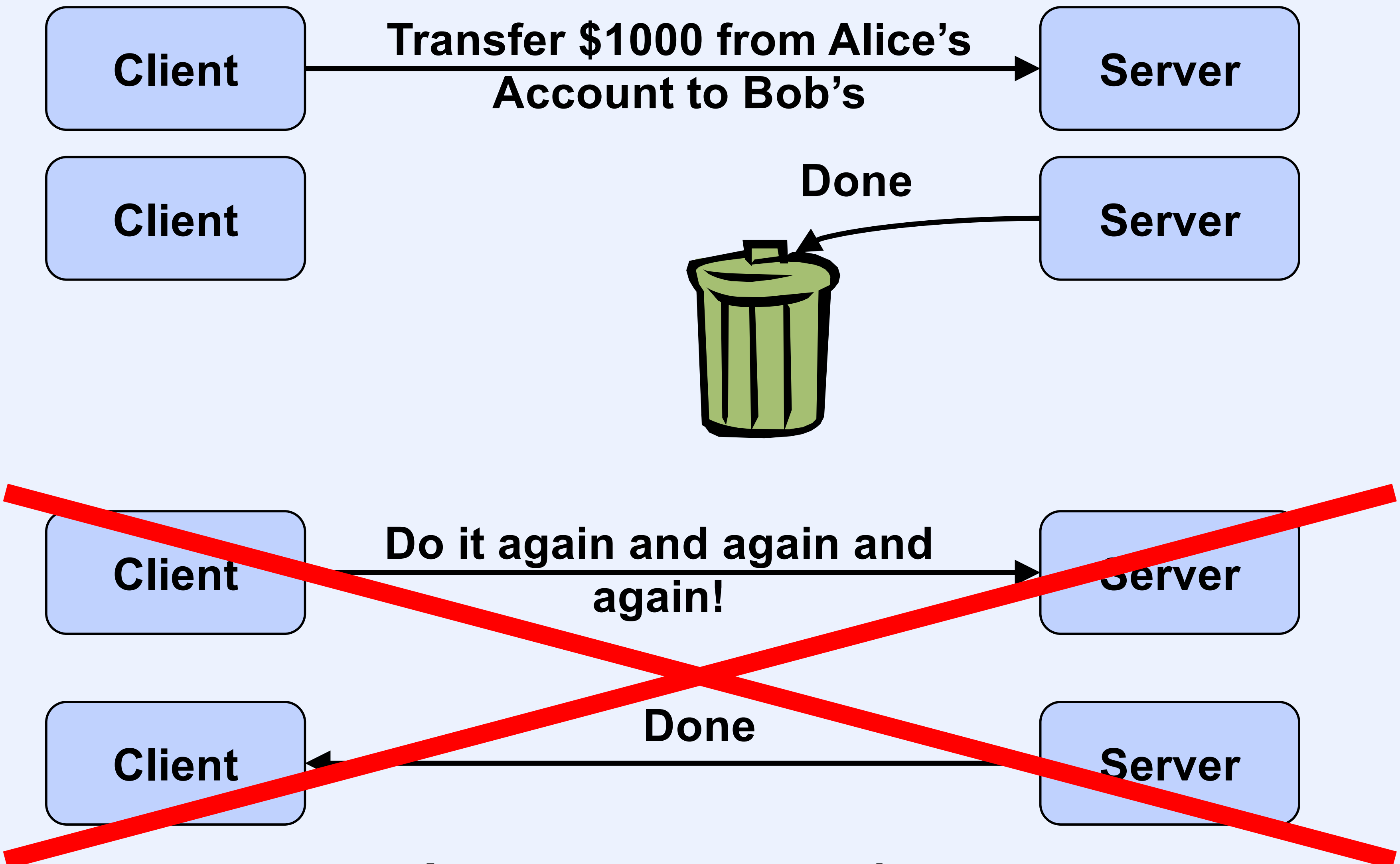


Idempotent Procedures



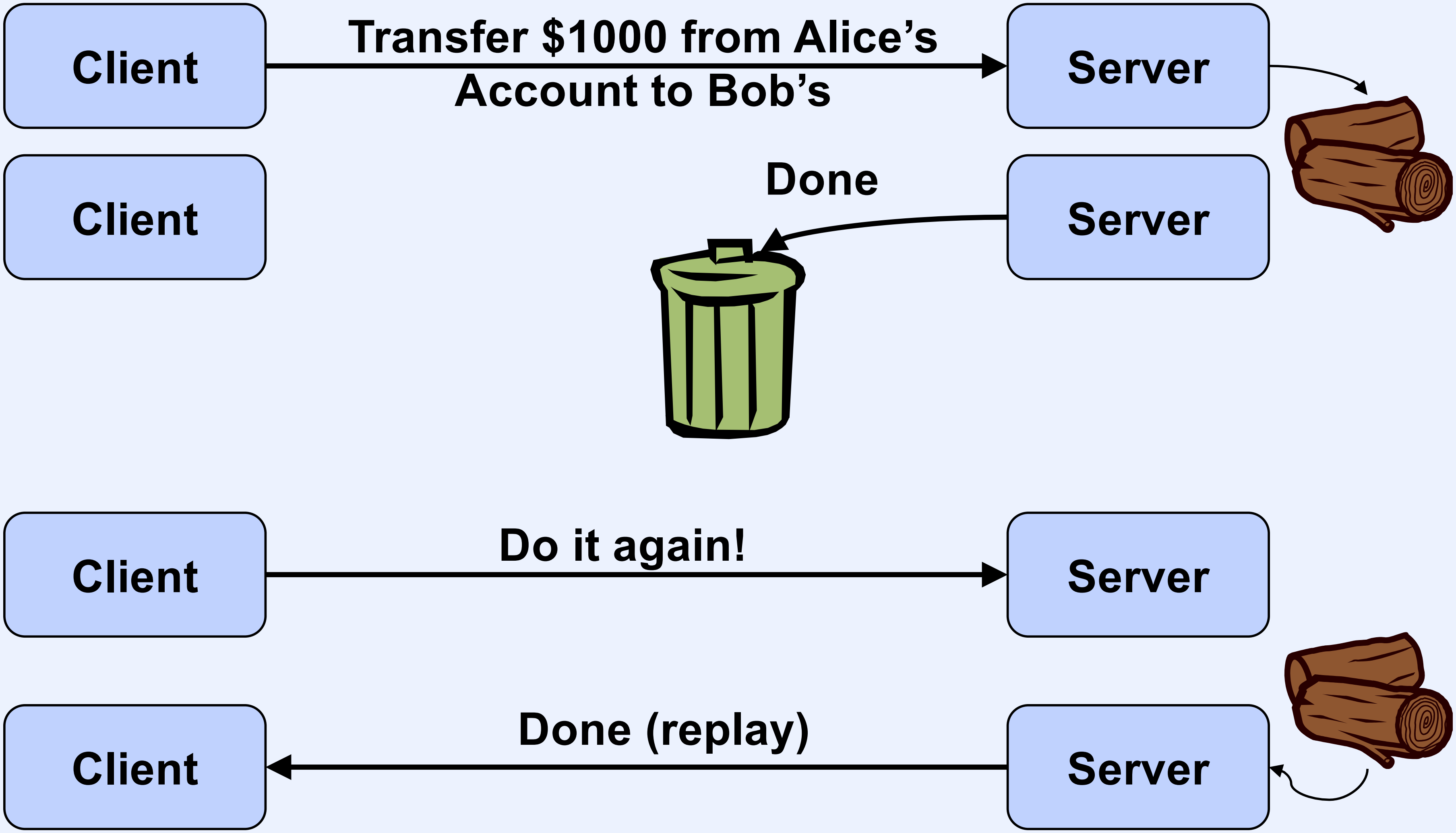
At-least-once **semantics**

Non-Idempotent Procedures



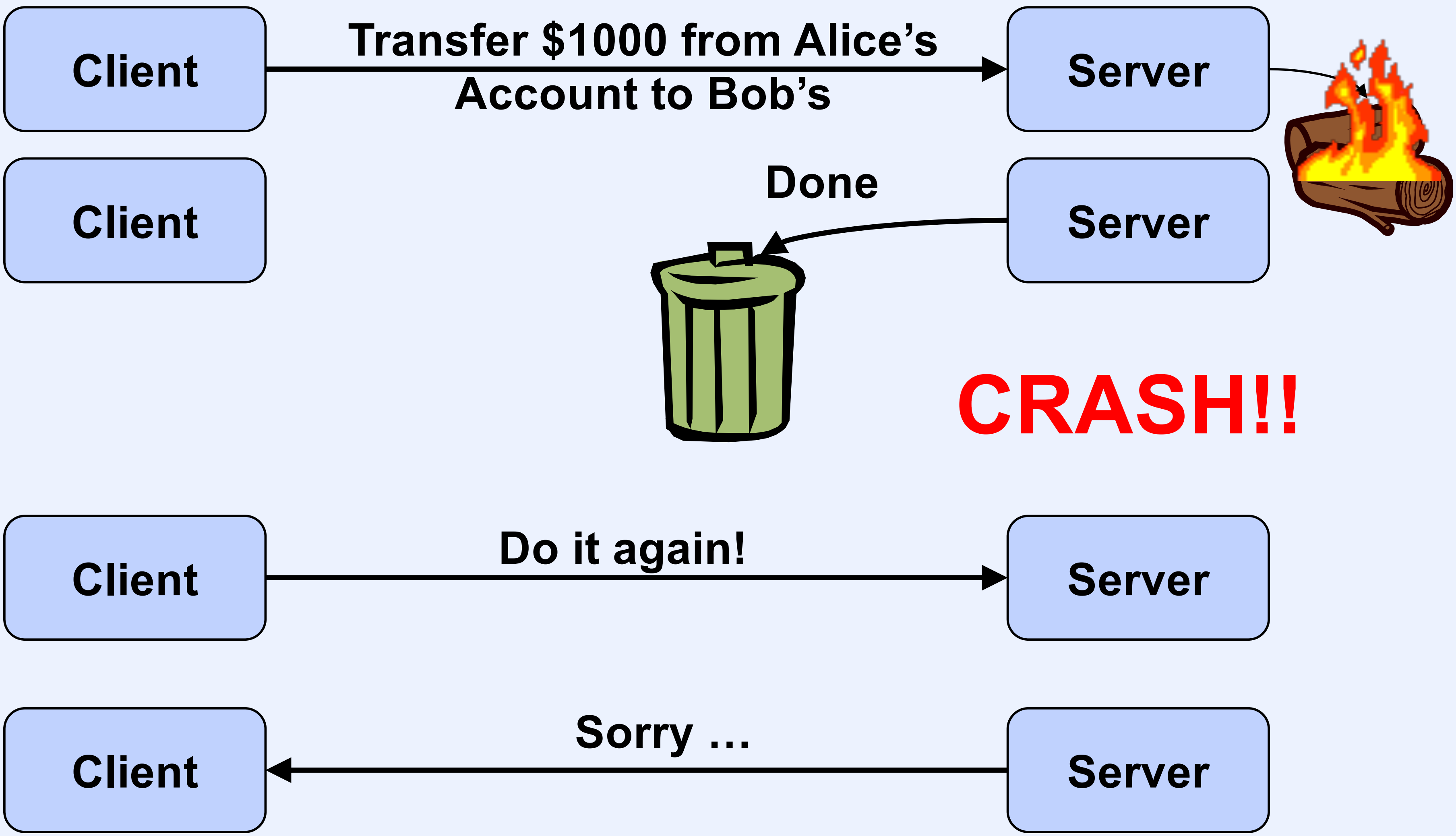
At-most-once semantics

Maintaining History



At-most-once semantics

No History

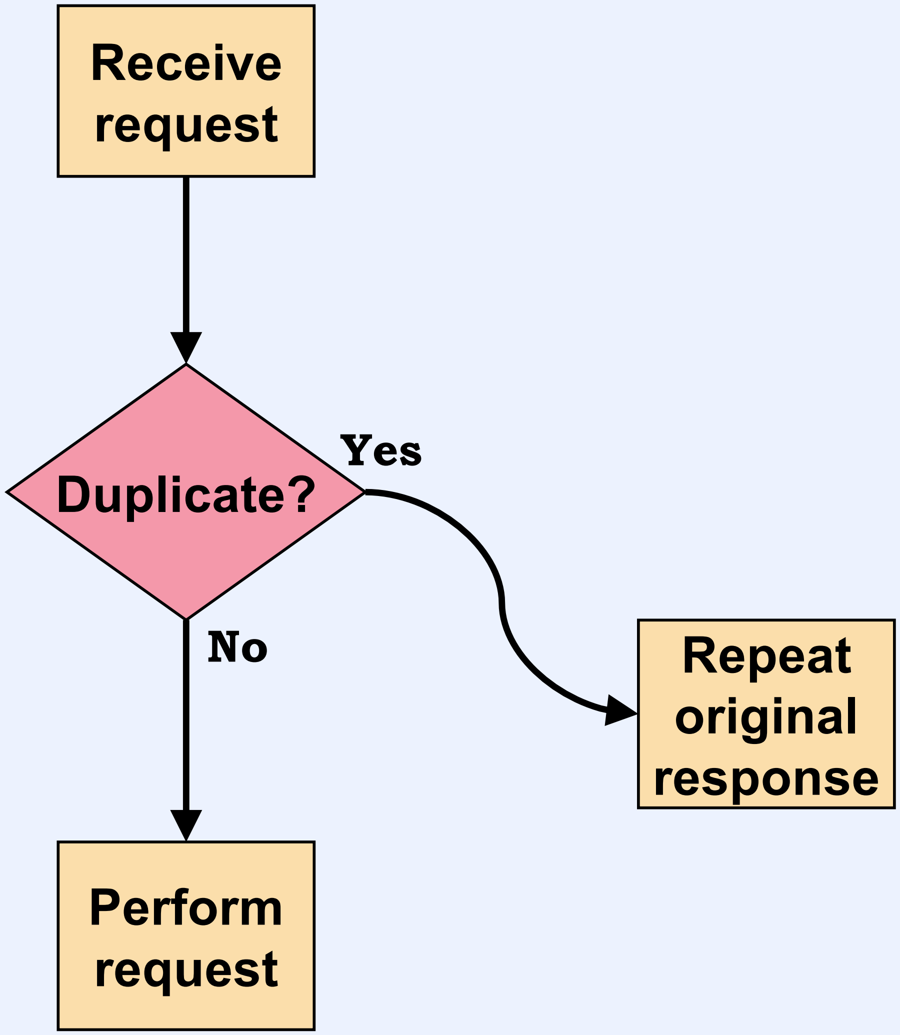


At-most-once semantics

Making ONC RPC Reliable

- **Each request uniquely identified by *transmission ID* (XID)**
  - **transmission and retransmission share same XID**
- **Server maintains *duplicate request cache* (DRC)**
  - **holds XIDs of *non-idempotent* requests and copies of their complete responses**
  - **kept in cache for a few minutes**

Algorithm

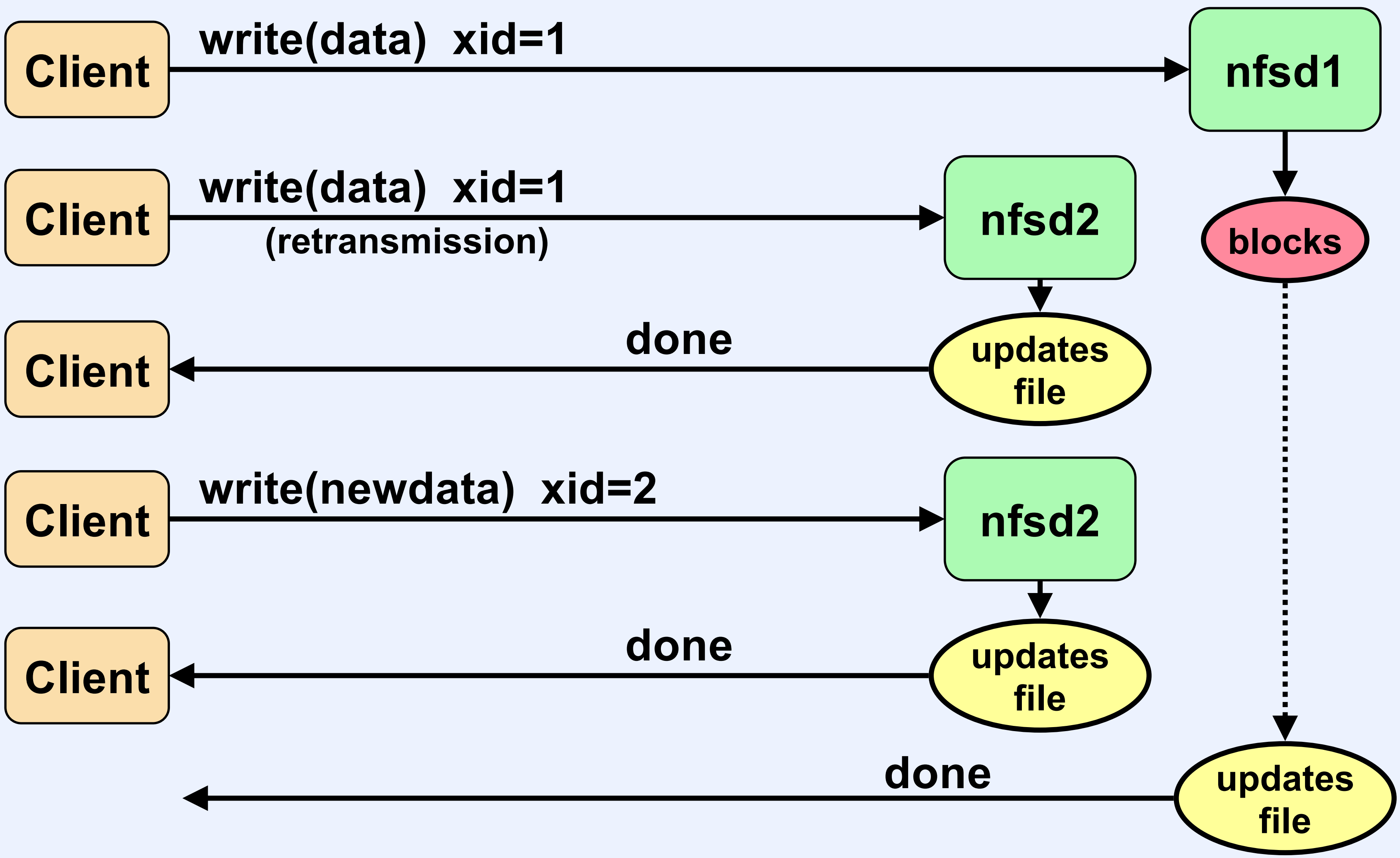


Receive request
Duplicate?
Yes
No
Repeat original response
Perform request

Did It Work?

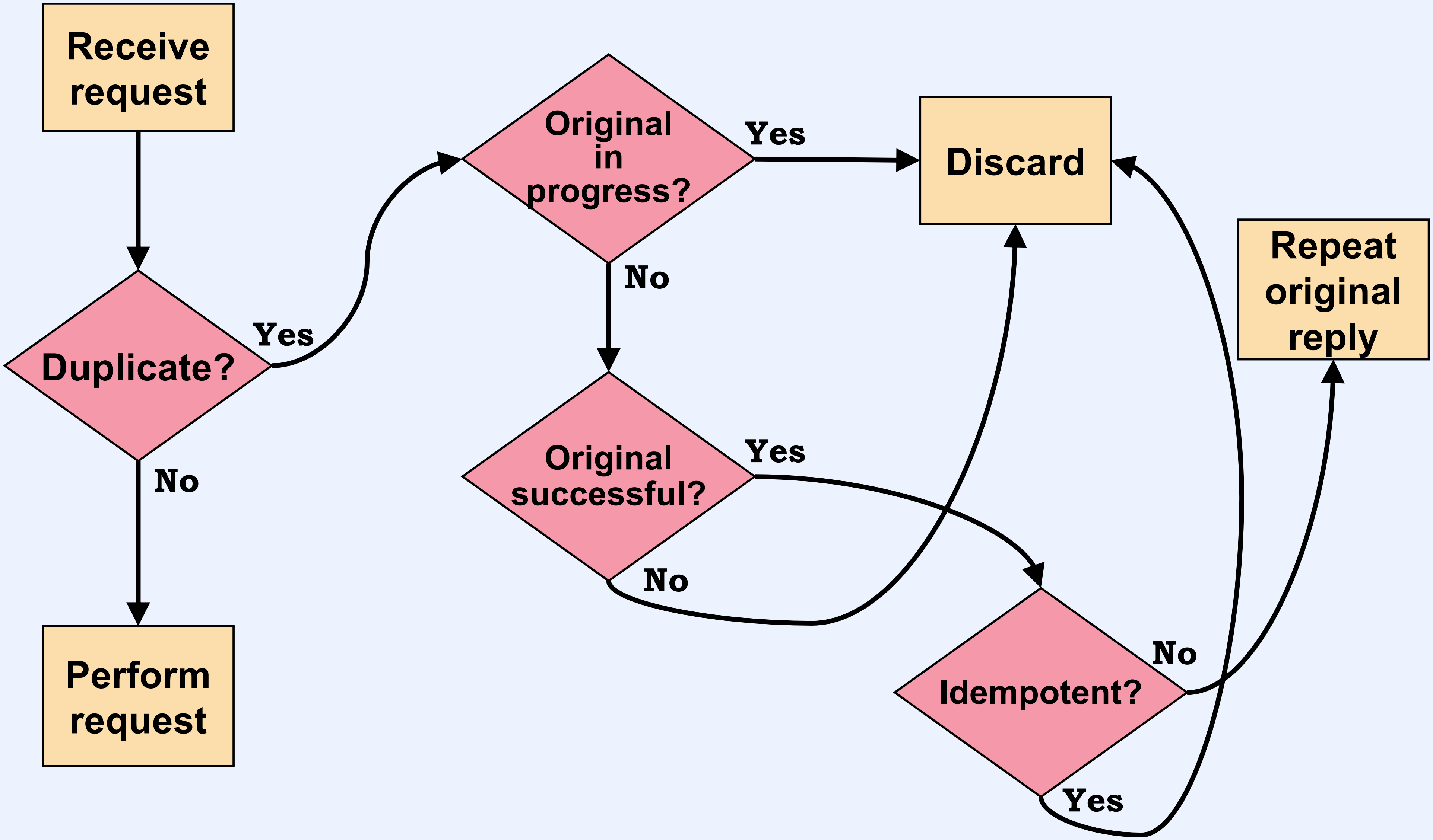
- No

Problem …



Client
write(data) xid=1
nfsd1
Client
write(data) xid=1
(retransmission)
nfsd2
blocks
Client
done
updates file
Client
write(newdata) xid=2
nfsd2
Client
done
updates file
done
updates file

Solution



Receive request
Duplicate?
Yes
No
Perform request
Original in progress?
Yes
No
Discard
Original successful?
Yes
No
Idempotent?
No
Yes
Repeat original reply

Quiz 2

An idempotent request from the client is received by the server, is successfully executed, and the response sent back. But the response doesn't make it to the client.

a) The client retransmits its request and the original response is sent back (again) to the client

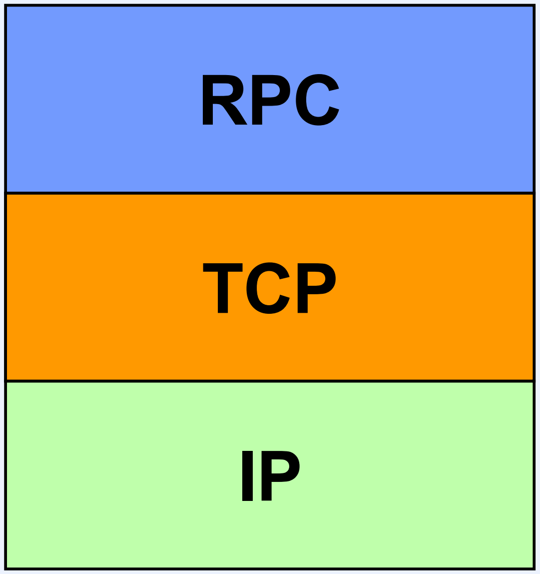
b) The client retransmits its request, but the original response is not sent back and thus, from the client's point of view, the server has crashed

c) The client, after multiple retransmissions, eventually gets a response

Did It Work?

- Sort of …
- Works fine in well behaved networks
- Doesn't work with "Byzantine" routers
  - programmed by your worst (and brightest) enemy
  - probably doesn't occur in local environment
  - good approximation of behavior on overloaded Internet
- Doesn't work if server crashes at inopportune moment (and comes back up)

Enter TCP



RPC as a Session Layer

- RPC is layered on the transport layer
- RPC “session” is a sequence of calls and responses
- If transport connection (if relevant) is lost, RPC creates a new one

Quiz 3

UDP is easy to implement efficiently. Early implementations of TCP were not terribly efficient, therefore early implementations of RPC were layered on UDP, on the theory that UDP usually provided reliable delivery.

a) TCP is reliable, therefore layering RPC on top of TCP makes RPC reliable

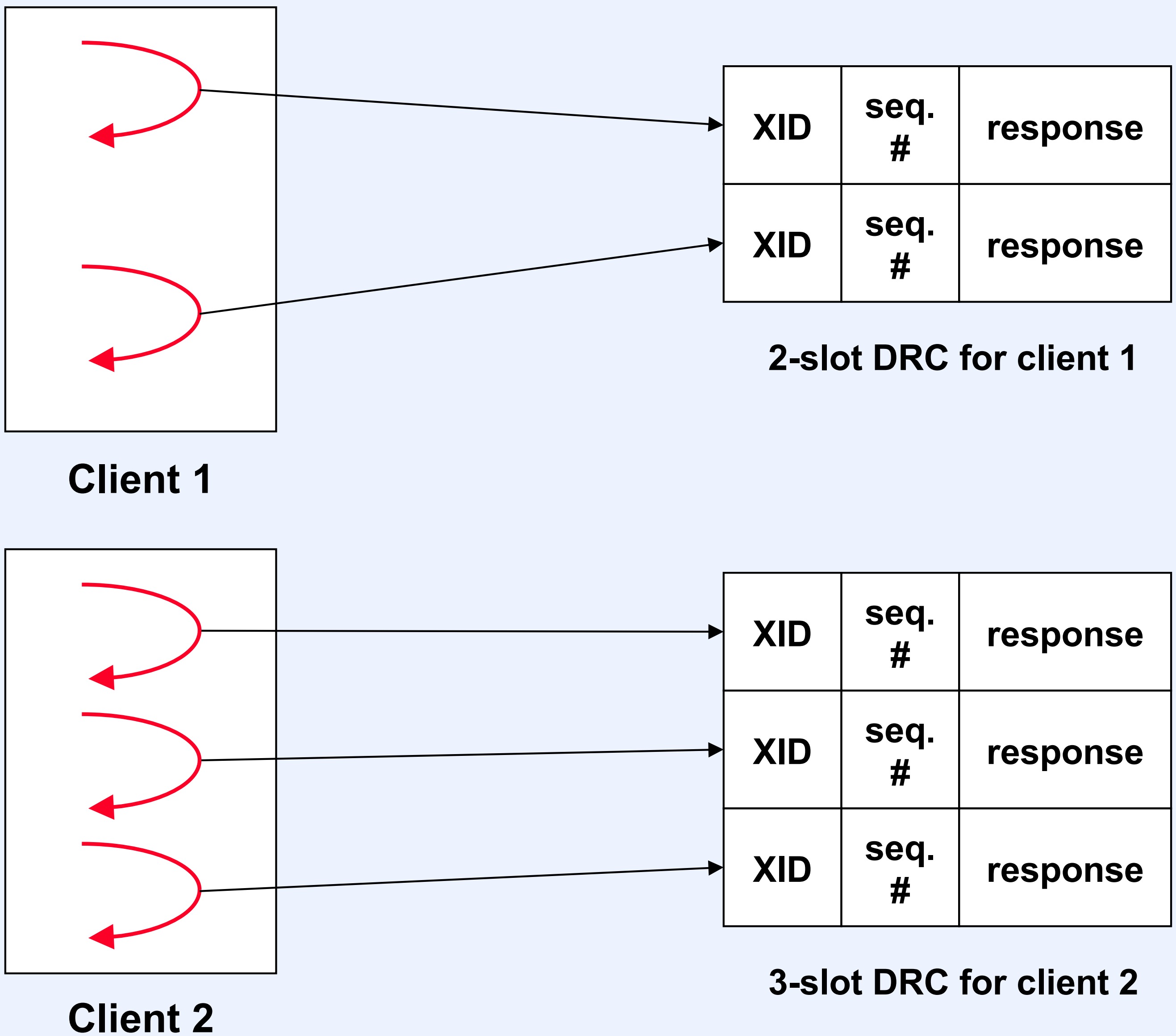
b) The notions of at-most-once and at-least-once semantics are still relevant, even if RPC is layered on top of TCP

c) There are additional reliability concerns, beyond those of UDP, when layering RPC on top of TCP

What's Wrong?

- The problem is the duplicate request cache (DRC)
  - it's necessary
  - but when may cached entries be removed?

Session-Oriented RPC

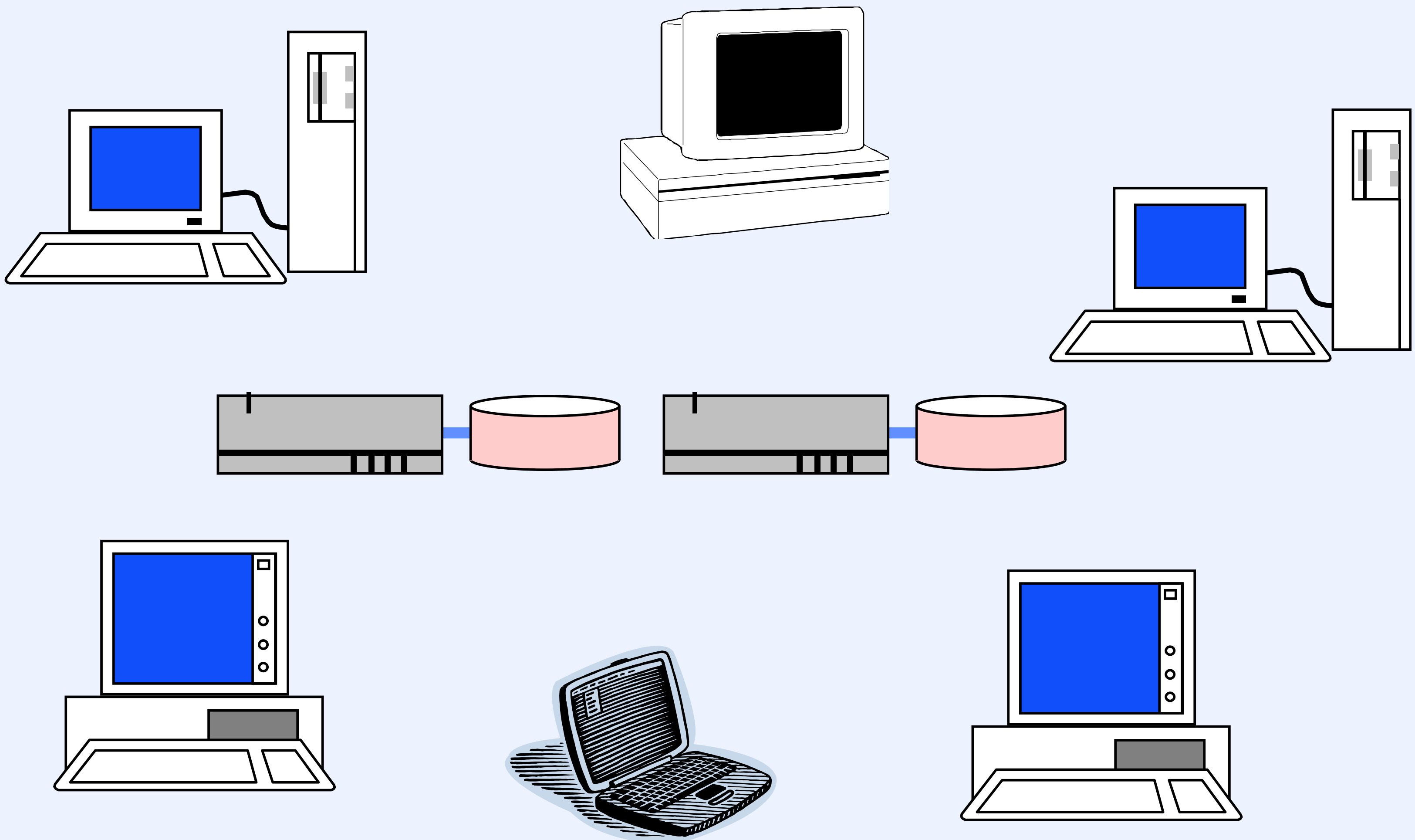


DCE RPC

- Designed by Apollo and Digital in the late 1980s
  - both companies later absorbed by HP
- Does everything ONC RPC can do, and more
- Basis for Microsoft RPC

Distributed File Systems Part 1

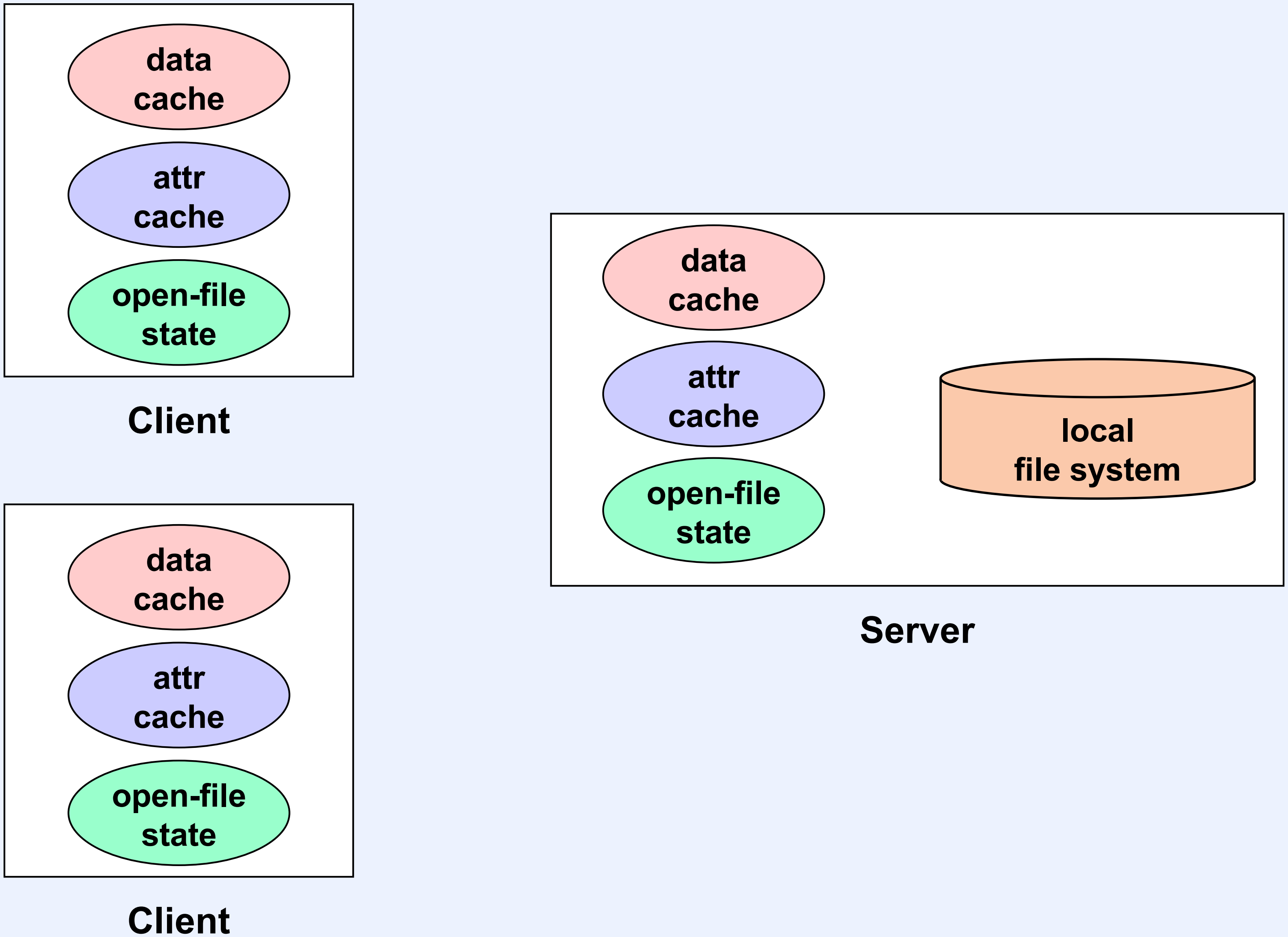
Distributed File Systems



DFS Components

- Data state
  - file contents
- Attribute state
  - size, access-control info, modification time, etc.
- Open-file state
  - which files are in use (open)
  - lock state

Possible Locations



Quiz 4

We'd like to design a file server that serves multiple Unix client computers. Assuming no computer ever crashes and the network is always up and working flawlessly, we'd like file-oriented system calls to behave as if all parties were on a single computer.

a) It can't be done

b) It can be done, but requires disabling all client-side caching

c) It can be done, but sometimes requires disabling client-side caching

d) It can be done, irrespective of client-side caching