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

A TCP receiver responds with an Ack to everything received.

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

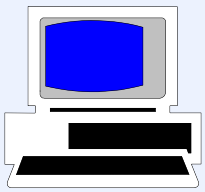
Fast Retransmit and Recovery

- 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)

Note that RTO stands for retransmission timeout–how long the system waits for an acknowledgement before giving up and retransmitting.

Remote Procedure Call Protocols

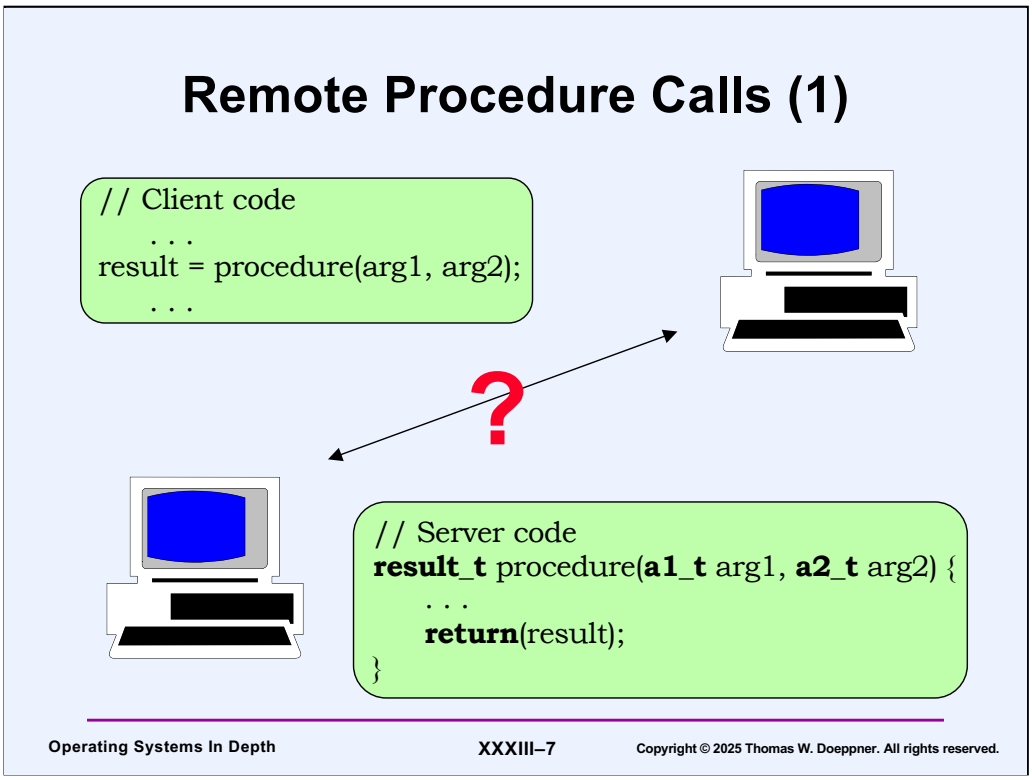
Local Procedure Calls



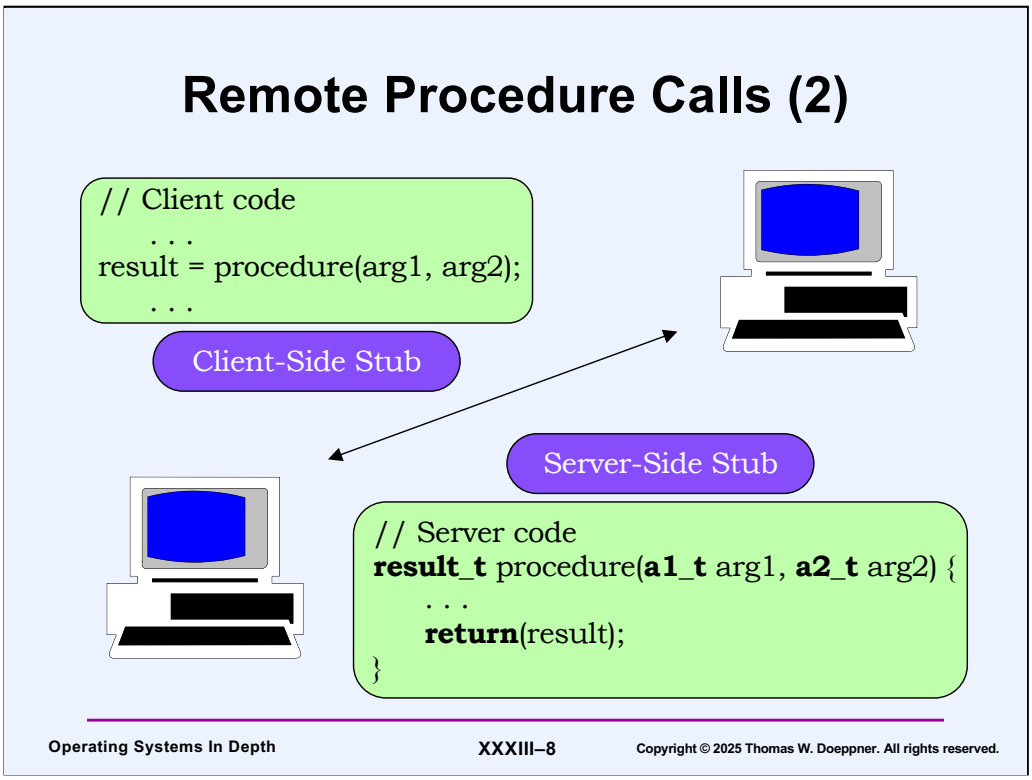
```
// Client code
    . . .
result = procedure(arg1, arg2);
    . . .
```

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


The basic theory of operation of RPC is pretty straightforward. But, to understand **remote** procedure calls, let's first make sure that we understand **local** procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a copy of the arguments (other languages, such as C++, provide other argument-passing modes, but copying is all that is provided in C). In some implementations, the callee's copy of the arguments have been placed on the runtime stack by the caller—the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be pointers—the callee might modify the value pointed to. The modified target of the pointer can then be seen by the caller on return from the callee.



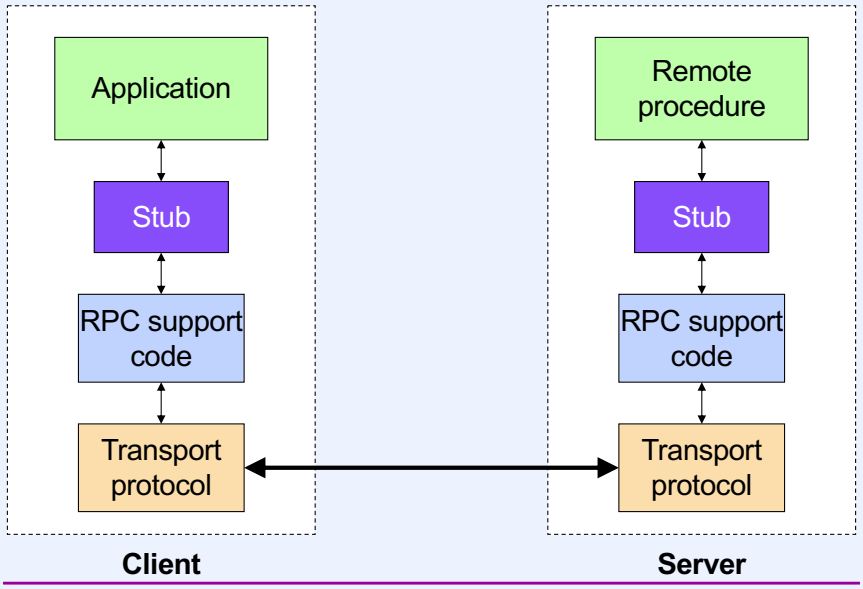
Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee might modify the target of a pointer and expect the caller to observe the modified value. These techniques simply don't work in the remote case.



The solution is to use **stub procedures**: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the **client-side stub**. This proxy gathers together all of the arguments and packages them into a message that it sends to the server. The server has a corresponding **server-side stub** that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. If the caller has modified the values pointed to by pointers, these modified values are also packaged up and sent back to the caller, where its stub copies the new values to the targets of the pointers.

From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call—they behave no differently for the remote case.

Block Diagram



ONC RPC

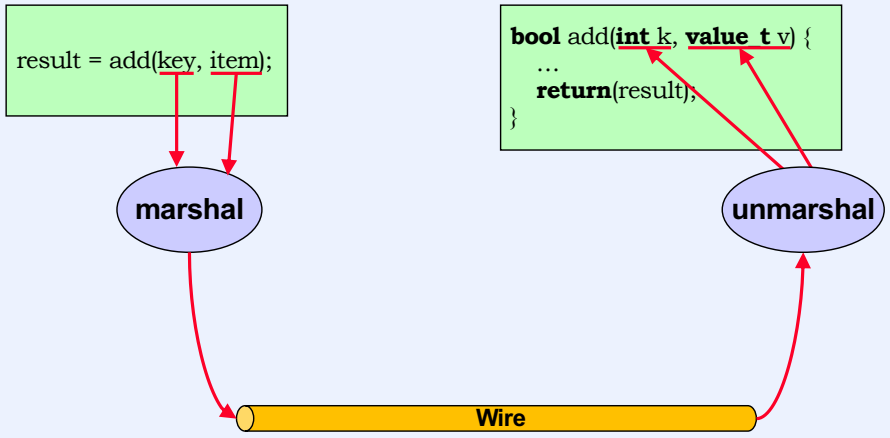
- Used with NFS
- eXternal Data Representation (XDR)
  - specification for how data is transmitted
  - language for specifying interfaces

ONC stands for open network computing and was originally designed by Sun Microsystems in the 1980s.

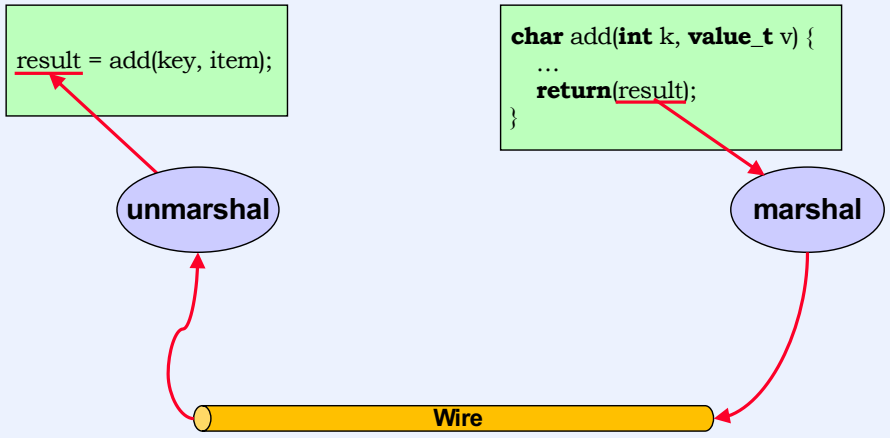
Example

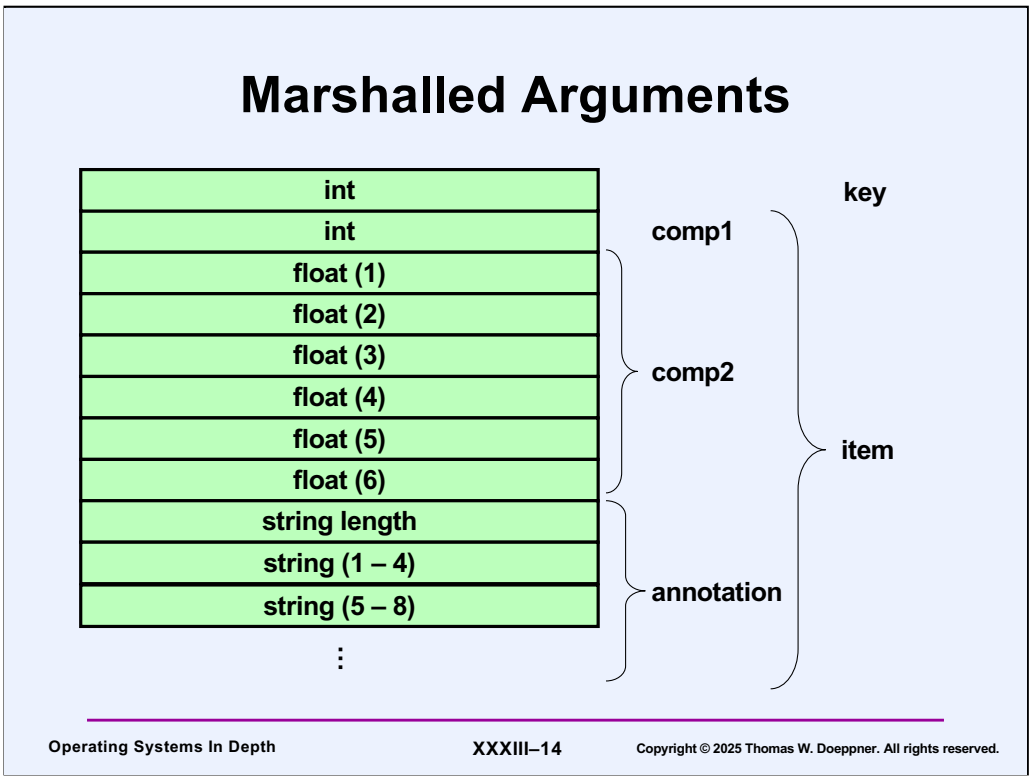
```
typedef struct {
   int   comp1;
   float comp2[6];
   char  *annotation;
} value_t;

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

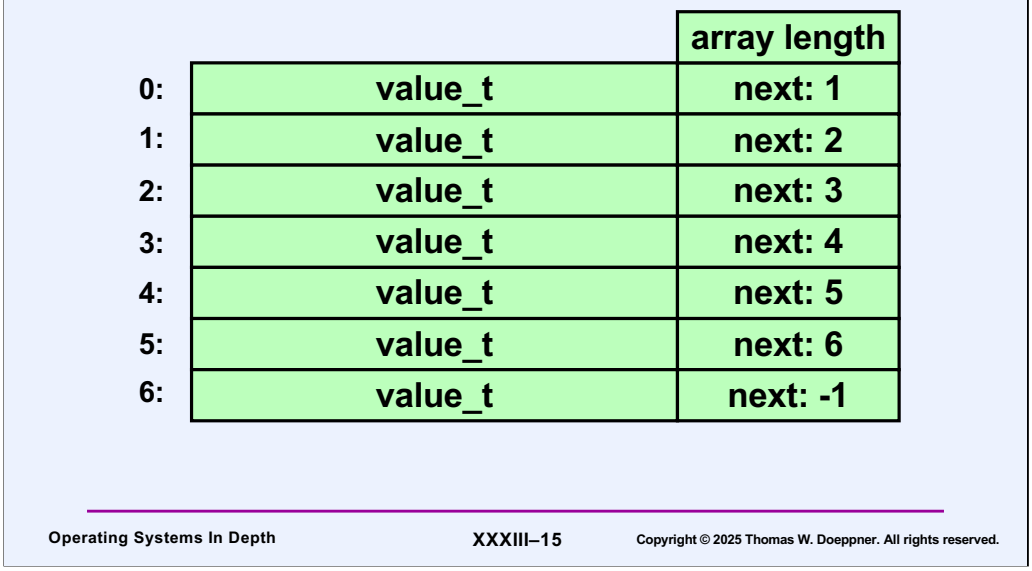




Here is what is sent over the wire after marshalling an item of type value_t.

Note that, as part of marshalling, the length of the string is determined and included with the marshalled arguments.

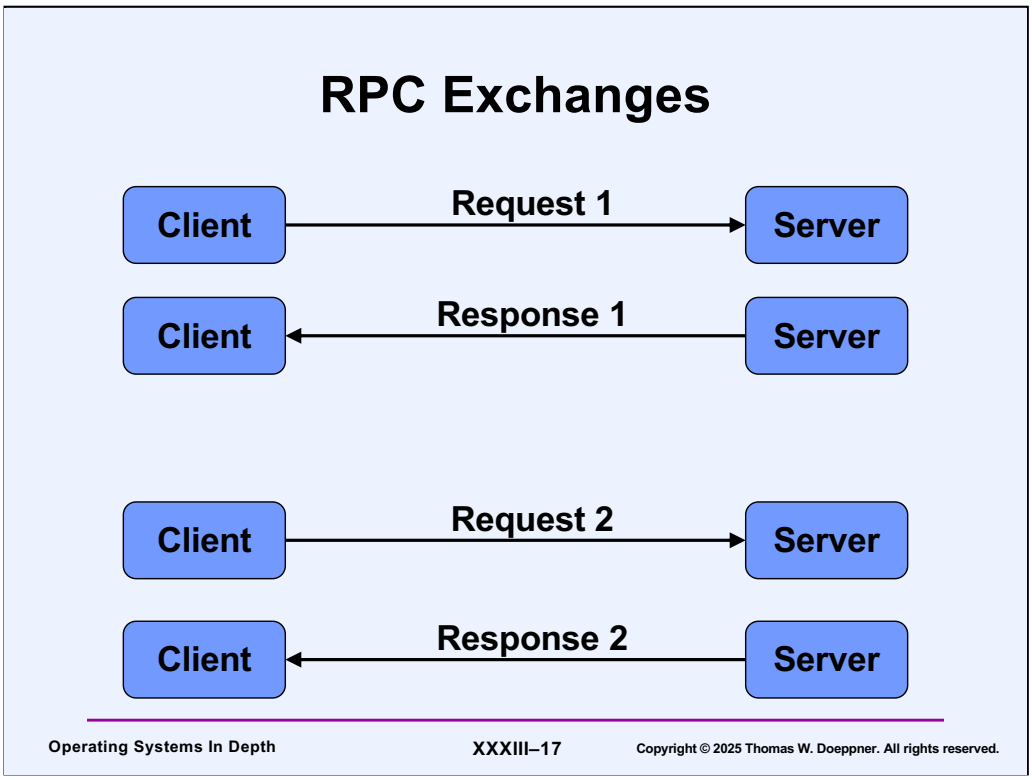
Marshalled Linked List



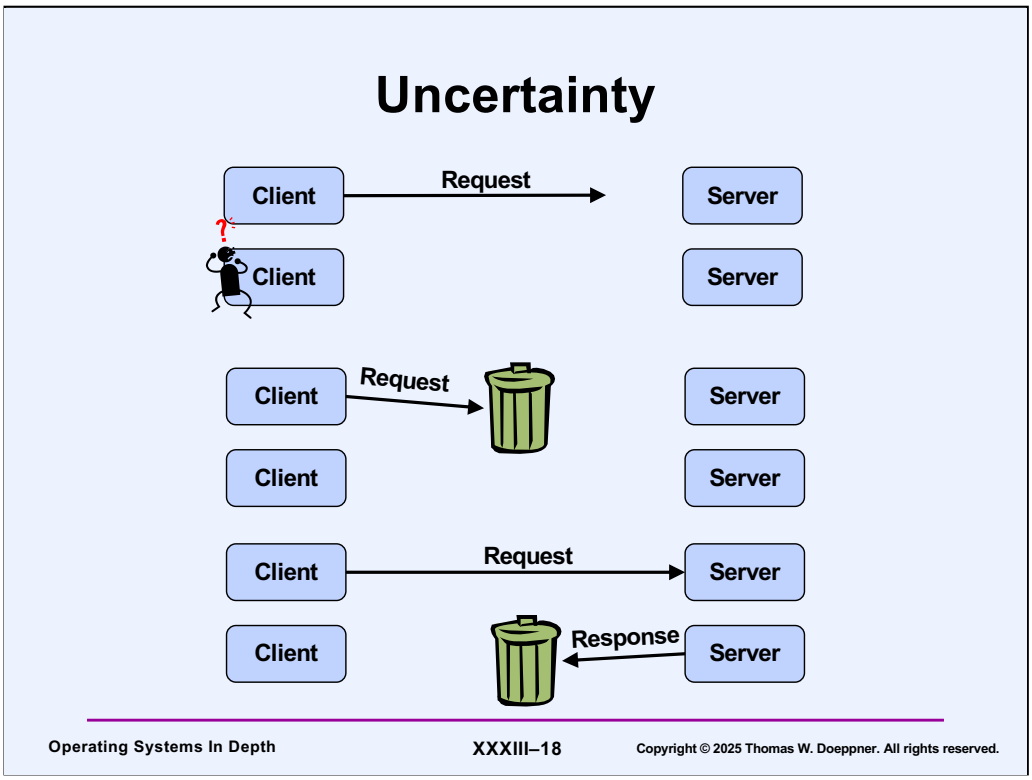
To marshal a list, one might represent it as an array with the links represented as array indices.

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

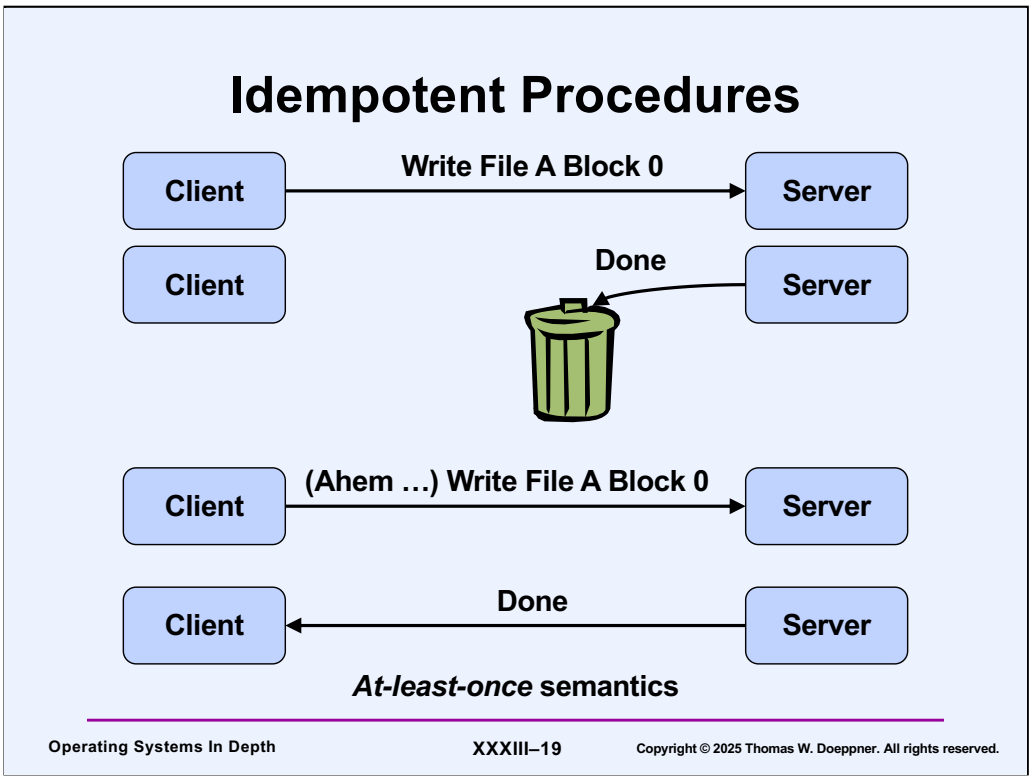


Note that if these exchanges are done over TCP, then each request and each response requires two packets—the message and then the ack. In UDP the response can be thought of as acknowledging the request, and the next request as acknowledging the response. Thus, in principle, RPC layered on UDP requires fewer messages than RPC layered on TCP.

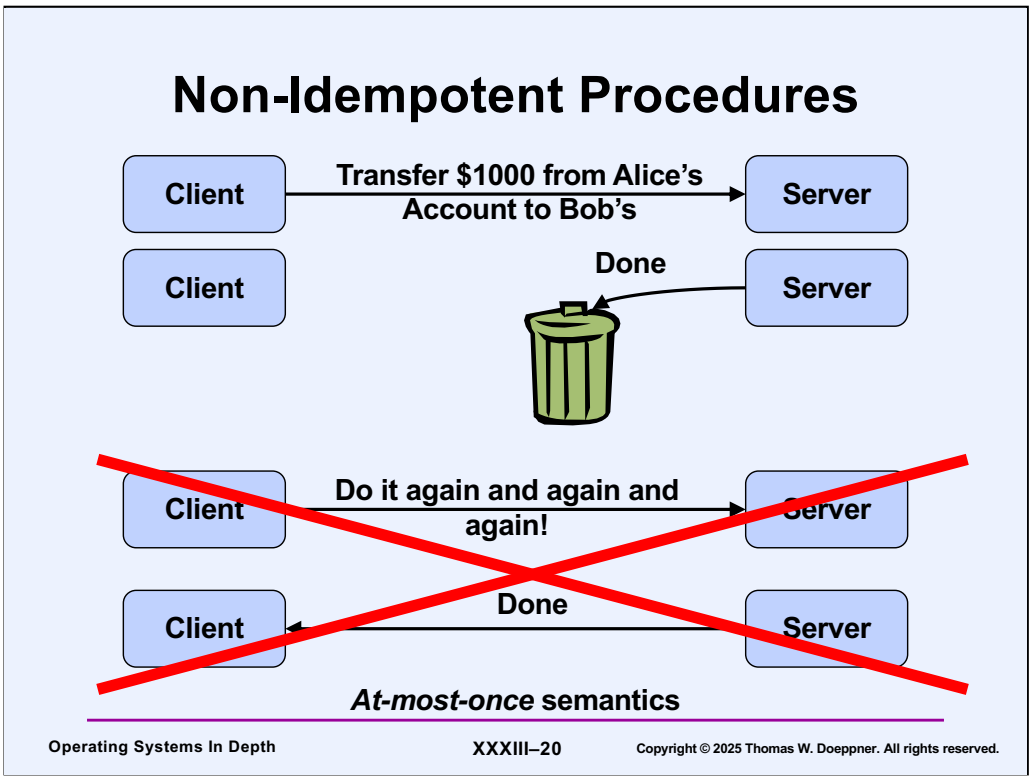


Consider this slide first with the assumption that RPC is layered on UDP. Thus, since the response acts as the acknowledgement, there is uncertainty as to whether the request was handled by the server.

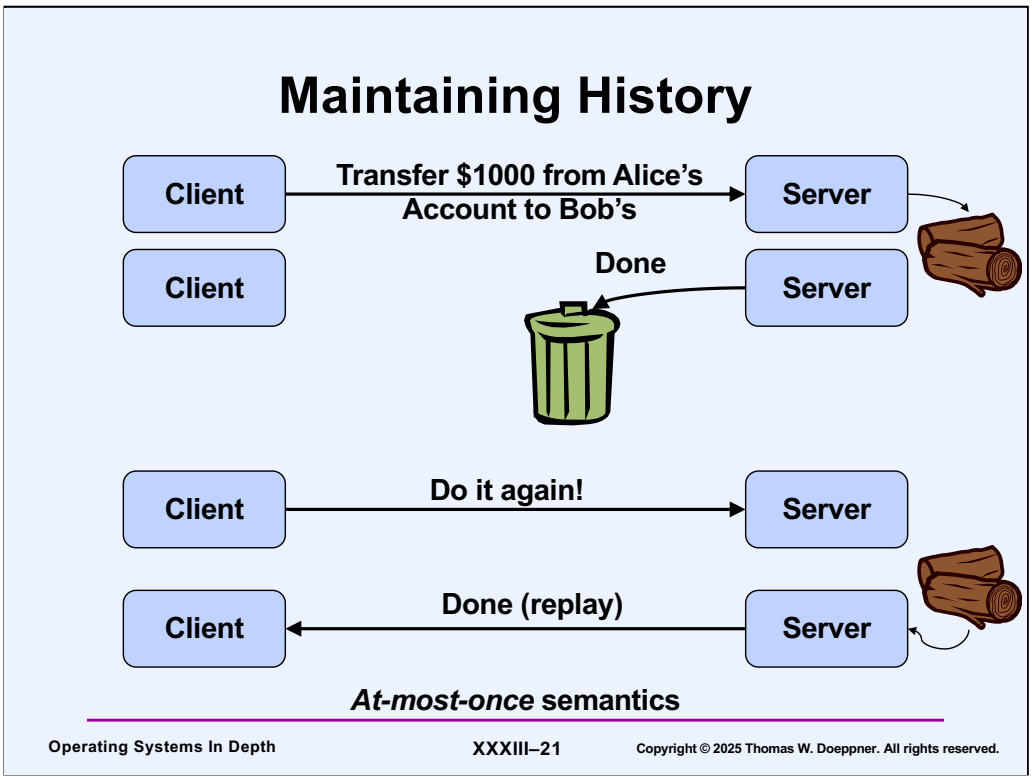
Does this uncertainty go away if RPC is layered on TCP? If you consider the possibility that the TCP connection might be lost, perhaps due to a transient network problem, the answer is clearly no. For example, suppose the TCP connection is lost just after the server receives the request. With no connection, the server cannot send a response, so the client is uncertain about what happened.



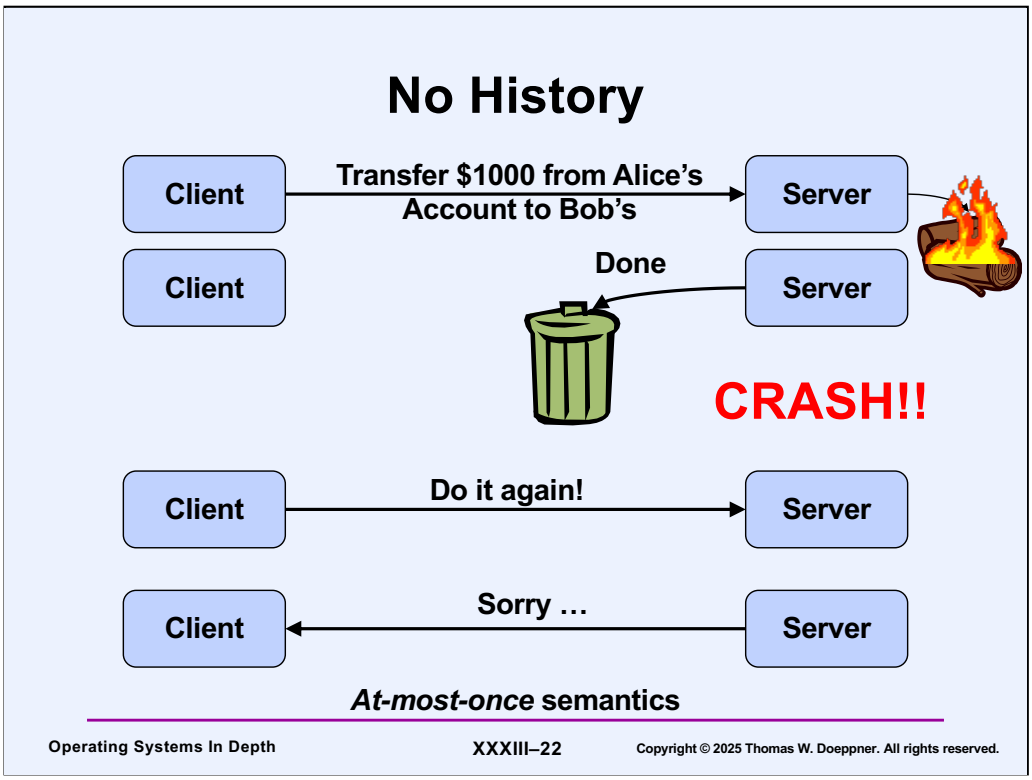
A procedure is **idempotent** if the effect of executing it twice in a row is the same as executing it just once. With such procedures, the client may repeatedly send a request until it finally gets a response. If an RPC protocol depends on such retries, it is said to have **at-least-once semantics**—clients are assured that, after all the retries, the remote procedure is executed at least once.



Not everything is idempotent! If we have non-idempotent procedures, then RPC requests should not be blindly retried, but instead should be sent just once. RPC protocols that do this are said to have **at-most-once semantics**.



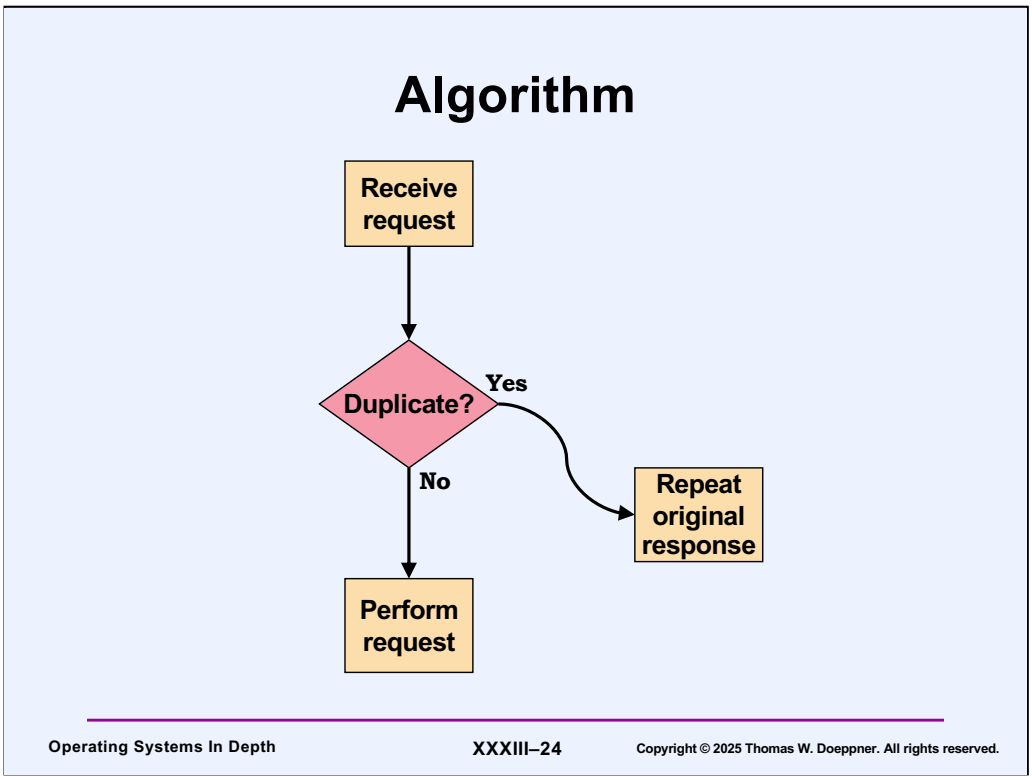
The server might keep track of what operations it has already performed and what the responses were. If it gets a repeat of a previous request, it merely repeats the original response.



If the server crashed and no longer has its history information, it can respond by raising an exception at the client, indicating that it has no knowledge as to whether the operation has taken place. But it guarantees that it hasn't taken place more than once.

Making ONC RPC Reliable

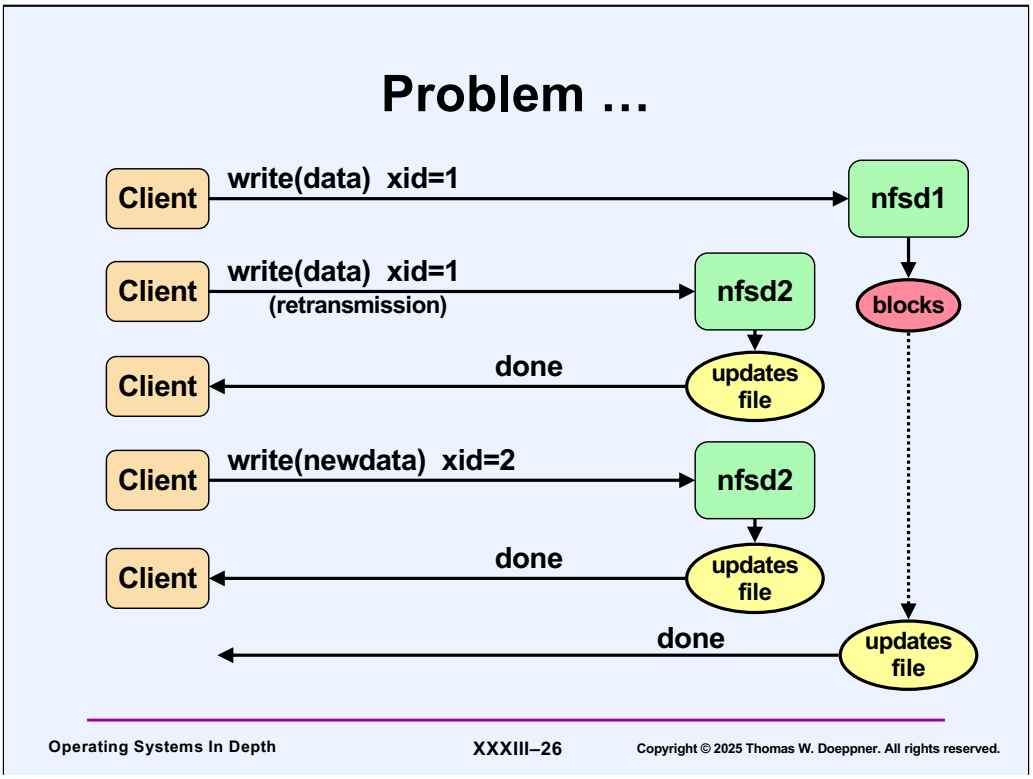
- Each request uniquely identified by *transmission ID* (XID)
  - transmission and retransmission share same XID
- Server maintains *duplicate request cache* (DRC)
  - holds XIDs of *non-idempotent* requests and copies of their complete responses
  - kept in cache for a few minutes



Note that idempotent requests are not cached and hence won't be recognized as duplicates.

Did It Work?

- No

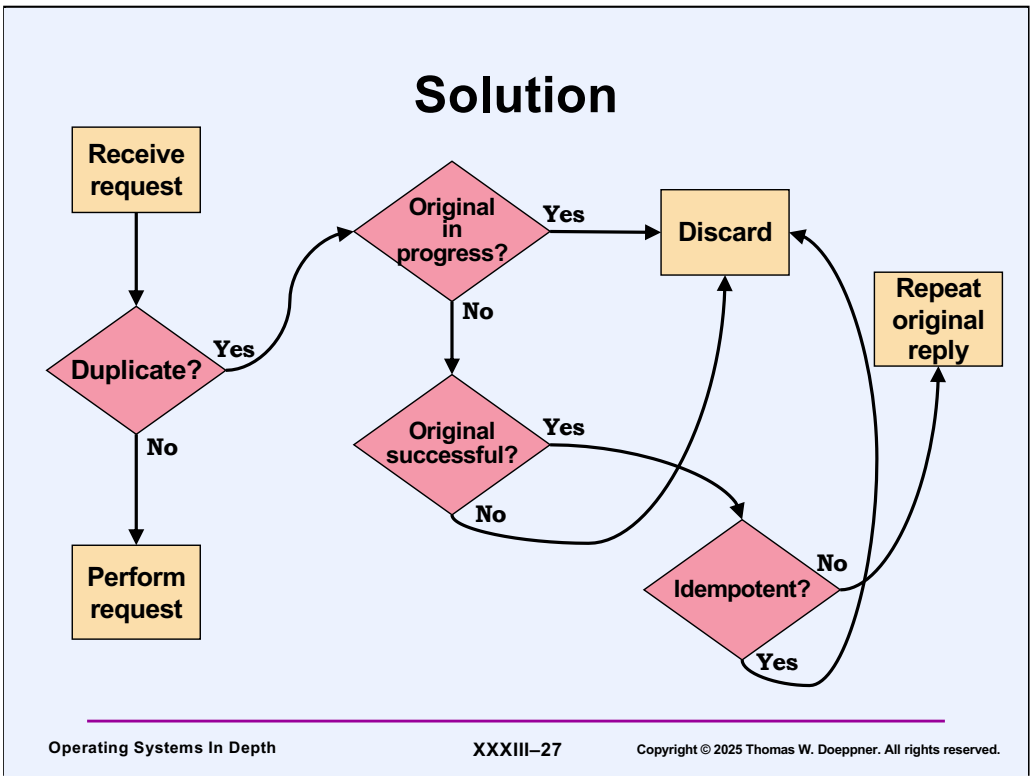


The server has a number of threads, known as nfsd's (NFS demons), that handle incoming requests.

In this example, the client sends a write request to the server, and nfsd1 handles it. However, nfsd1 must wait for something to be available and blocks (the server is probably overloaded). Note that write requests, which specify both file and location within file, are idempotent and thus not cached in the DRC.

The client times out and retransmits the request, which is handed by nfsd2. Nfsd2 does not have to wait and quickly updates the file, and respond to the client. The clients now sends another request, also handled by nfsd2, to write a new value in the same file location as used by the previous request. This request is speedily dealt with.

Nfsd1 finally wakes up and updates the file, overwriting what was just written by nfsd2. Thus the second update to file is replaced with the first update.



To deal with the problems of the previous slide, the server's algorithm was modified as shown in this slide. Furthermore, all requests are cached in the DRC, not just non-idempotent ones.

Some of the paths through this flow chart might seem a bit odd. First, note that requests remain in the duplicate request cache for a relatively short amount of time. It's assumed that the the normal cause of a retransmission is that the sender retransmitted too soon—either that the original request is still in progress or that the original response has already been sent. If this assumption is not correct (for example the response to an idempotent request was sent but it never made it to the client), the sender may have to retransmit several times before the original request times out of the cache, thus allowing the retransmission to be treated as a new request.

It's not at all clear why, if the original request was not successful, that the retransmission is discarded. The presenter of the paper describing the approach said that original algorithm did that, so they decided to mimic what it did in this version so as not to break anything.

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

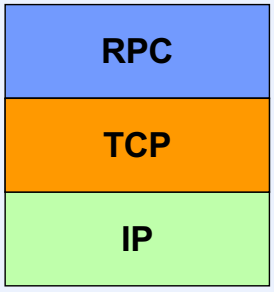
c) The client, after multiple retransmissions, eventually gets a response

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)

An example of a situation in which it wouldn't work is if a non-idempotent request is sent to the server. The server performs the request and sends back a response, but the response is lost. The client repeats the request but waits too long to do so: the original request times out of the DRC. When the repeated request arrives at the server, the server considers it a new request and performs it (again).

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.

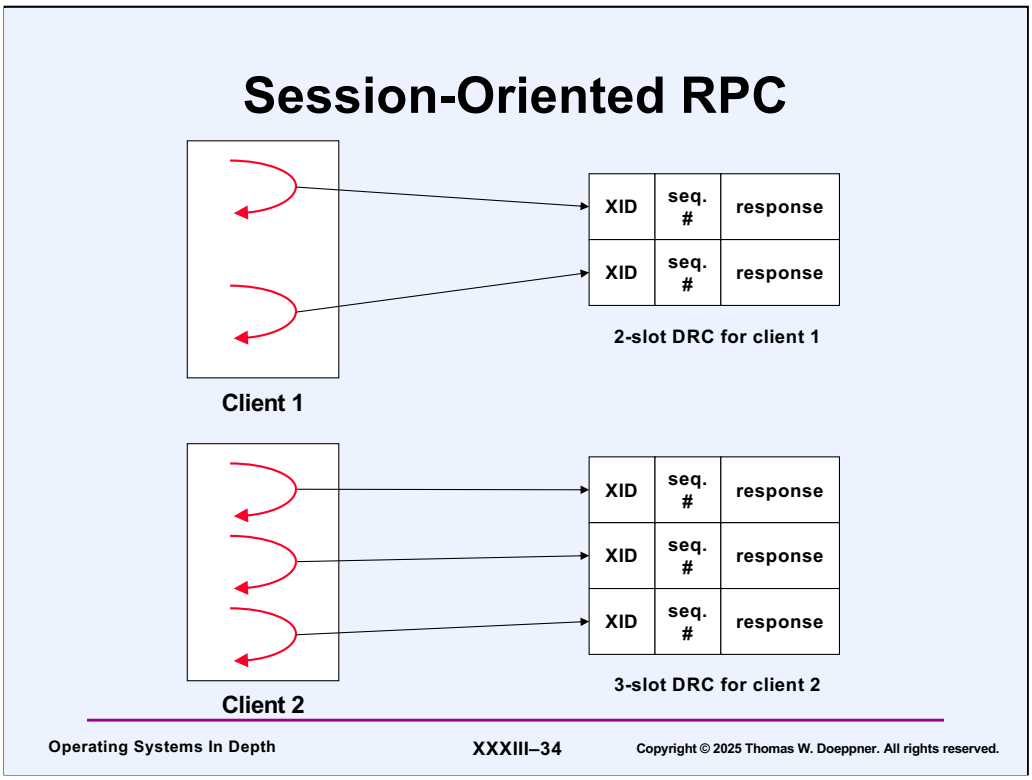
a) TCP is reliable, therefore layering RPC on top of TCP makes RPC reliable

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?



In session-oriented RPC, the client, as part of setting up a **session** with the server, lets the server know ahead of time the maximum number of concurrent requests it will issue; the server then creates that number of **channels** for it. Each client request then provides a channel number and a sequence number. Only one request at a time may be active on a channel. Requests on each channel carry consecutive sequence numbers. The server maintains a separate DRC entry for each channel of each client, containing the XID, sequence number, and response, as shown in the slide. An entry's contents are not deleted until a request arrives with the next sequence number for that channel.

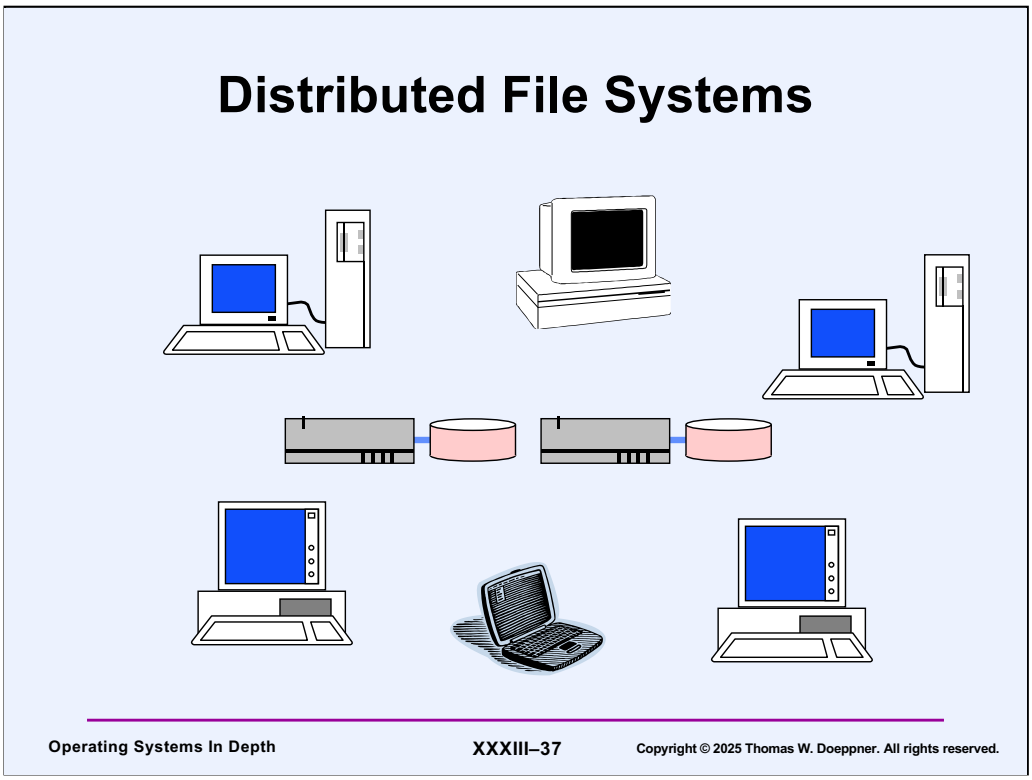
DCE RPC

- 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

To be precise, Digital Equipment Corporation (DEC) was purchased by Compaq in 1998, which was acquired by HP in 2002. Apollo was acquired by HP in 1989.

The principal software architect of DCE RPC joined Microsoft after the sale of their company and led the design of Microsoft RPC.

Distributed File Systems Part 1

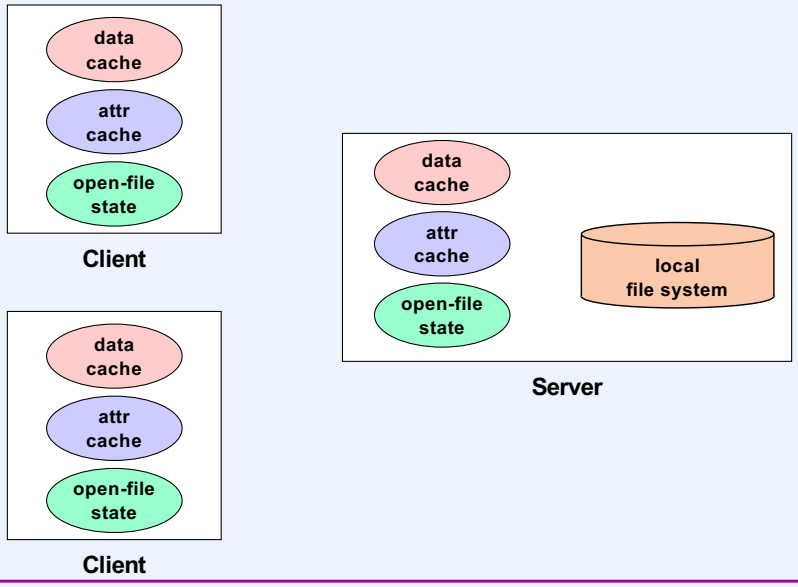


File systems are certainly important parts of general-purpose computer systems. They are responsible for the storage of data (organized as files) and for providing a means for applications to store, access, and modify data. Local file systems handle these chores on individual computers; distributed file systems handle these chores on collections of computers. In the typical design, distributed file systems provide a means for getting at the facilities of local file systems. What is usually desired of a distributed file system is that it be access transparent: programs access files on remote computers as if the files were stored locally. This rules out approaches based on explicit file transfer, such as the Internet's FTP (file transfer protocol) and Unix's rcp and scp (remote copy).

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