

18. Remote Procedure Calls

We start by observing a virtual usage of the client/server approach

X PROTOCOL



The X server handles the keyboard, mouse, and display of the system

This is an example of a distributed system & uses remote procedure calls

- our goal is modularity rather than virtualization
- the server acts as the kernel

This has a few distinction from standard procedure calls

- i. the caller and callee do not need to share an address space
- ii. passing by reference & pointers are not allowed
- iii. the client & server need not be on the same architecture
 - a. i.e. x86 & x86-64 or x86-64 & SPARC64

But (iii) introduces long size and endian-ness

problems

- we address these by marshaling commands
 - this adds another layer of abstraction on the commands
 - bit pattern forms: XML (slow), JSON, IEE 754



What could go wrong?

- messages can get lost/duplicated/corrupted
- the network/server can go down (or be slow)

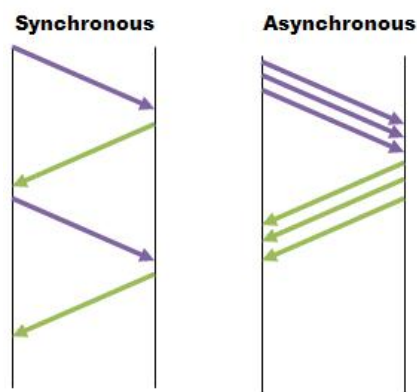
& these appear identical to the client!

We deal with these issues as two categories:

1. Corruption
 - a. better checksums in messages
 - b. end-to-end (rather than link) encryption
 - i. link encryption: devices decrypt and re-encrypt
 - ii. end-to-end encryption: data stays encrypted for the entire journey
 - c. resend messages on detection
2. Network Issues
 - a. At-Least-Once RPC (keep trying)
 - i. good for idempotent information
 - b. At-Most-Once RPC (log on error)
 - i. good for transactions
 - c. Exactly-Once RPC (don't make mistakes)
 - i. very (some would say too) difficult to implement

Another downside: INEFFICIENCY

We have 3 major approaches to combatting this:



1. Asynchronous System Calls (Pipelining)

- allows for more useful work time
- complicated partial failure
- out-of-order responses
- requests must be independent

2. Cache-on-the-Client

- cache responses to likely requests
- common with web browsers

3. Pre-Fetching

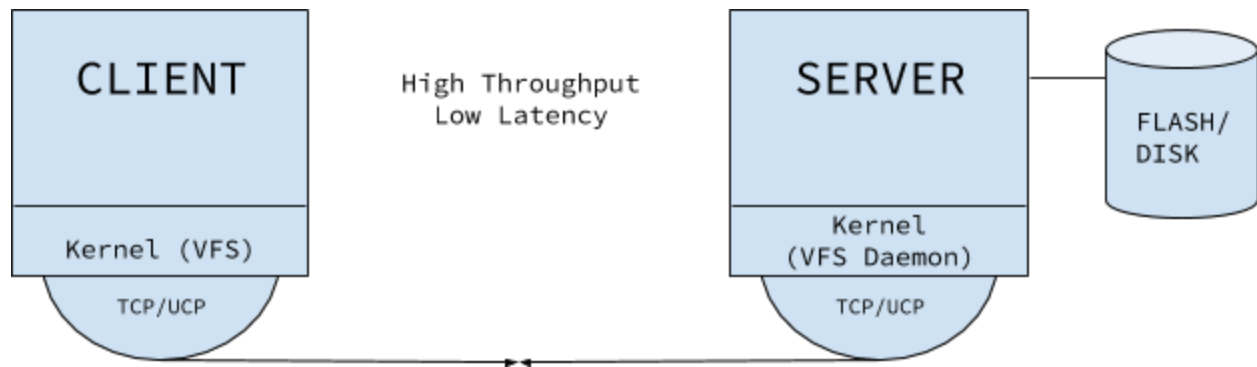
- guess a response prior to receiving methods
- hogs resources

We can use these ideas to implement

Networked File Systems

We analyze NFS models

- a Linux-type system which sends requests (like open, read, lseek...) to a network
 - v3: useful in machine rooms, since it uses UDP
 - UDP establishes a connection before sending packets
 - v4: useful for the internet, since it uses TCP
 - TCP sends packets without establishing a connection



The kernel uses a VFS (such as btrfs, ext4, etc) to

- transparently marshal procedure calls
- to send and receive unencrypted packets
 - encryption started with VFSv4

Basically, we emulate a local file system with a network!

This is great for modularity (we don't even need the same underlying architecture, remember?) but now we are relying on a network; what if it goes down?

- ex) read is short enough to avoid failure by interrupt; what if the network goes down?
 - Option A: read() hangs
 - this exposes us to infinite hangs!
 - Option B: send a ^C to interrupt
 - apps which assume a return from read get complicated!
- The system administrator has to choose, but we're wrong either way!

So what do we do?

In short, don't let the network go down!

But we design our protocol carefully just in case

Even without a crash, close() is costly, as the client waits for all outstanding requests

NFS Primitives:

```
# all elements are fixed-size
# fh and attr are concatenated and cast to integers
MKDIR(dirfh, name, attr) -> fh & sttr
REMOVE(dirfh, name)-> status
```

```
READ(fh, offset, size) -> data
LOOKUP(dirfh, name) -> fh & attr
# example message:
LOOKUP("/usr/local", "/bin") -> 4728 + {output of ls}
```

An nfs file handle uniquely identifies a file on a server

- it survives renames & reboots
- on Linux, it is a concatenation of device number, inode number, and serial number

They cause the server to be stateless, as they do not rely on the state of the client

So will a concurrent WRITE and RENAME cause problems?

~ NO; the file handle does not change

Will a concurrent REMOVE and WRITE?

~ YES, which violates the Linux standard! (would wait to delete, but NFSv3 is stateless)

There are two cases:

1. REMOVE & WRITE are from the same client
 - a. save a dummy request to a temp file names ".NFS#"
 - b. the client is responsible for the cleanup
2. REMOVE & WRITE
 - a. set errno = ESTALE

A related error occurs with concurrent REMOVE & WRITE or WRITE

~ the NFS would not recognize the stale request and would write to the wrong file
→ we need a serial number to represent what version of this file handle a file is