sockets con't / RPC

last time

client/server versus peer-to-peer model

names, addresses (IPv4/IPv6), routing

socket abstraction — two-way pipes to remote machine

server sockets

- bind() (set address) + listen() (wait for connections)
- accept() to create new connection socket

client sockets

- connect() to connect

getaddrinfo() — names to address

incomplete writes

`write` might write less than requested

- error after writing some data
- if blocking disabled with `fcntl()`, buffer full

`read` might read less than requested

- error after reading some data
- not enough data got there in time

handling incomplete writes

```
bool write_fully(int fd, const char *buffer, ssize_t count) {
    const char *ptr = buffer;
    const char *end = buffer + count;
    while (ptr != end) {
        ssize_t written = write(fd, (void*) ptr, end - ptr);
        if (written == -1) {
            return false;
        }
        ptr += written;
    }
    return true;
}
```

on filling buffers

```c
char buffer[SIZE];
ssize_t buffer_used = 0;

int fill_buffer(int fd) {
    ssize_t amount = read(
        fd, buffer + buffer_used, SIZE - buffer_used
    );
    if (amount == 0) {
        /* handle EOF */ ???
    } else if (amount == -1) {
        return -1;
    } else {
        buffer_used += amount;
    }
}
```

reading lines

(note: code below is not tested)

```
int read_line(int fd, const char *p_line, size_t *p_size) {
    const char *newline;
    while (1) {
        newline = memchr(buffer, '\n', buffer_used);
        if (newline != NULL || buffer_used == SIZE) break;
        fill_buffer();
    }
    memcpy(p_line, buffer, newline - buffer);
    *p_size = newline - buffer;
    memmove(newline, buffer, buffer + SIZE - newline);
    buffer_end -= (newline - buffer);
}
```

aside: getting addresses

on a socket fd: `getsockname` = local addresss

- `sockaddr_in` or `sockaddr_in6`
- IPv4/6 address + port

on a socket fd: `getpeername` = remote address

addresses to string

can access numbers/arrays in `sockaddr_in`/`in6` directly

another option: `getnameinfo`

- supports getting W.X.Y.Z form or looking up a hostname

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR “socket option”

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn’t check for reading multiple lines at once

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR "socket option"

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn't check for reading multiple lines at once

reading and writing at once

so far assumption: alternate between reading+writing

- sufficient for FTP assignment
- how many protocols work

“half-duplex”

don’t have to use sockets this way, but tricky

threads: one reading thread, one writing thread *OR*

event-loop: use *non-blocking I/O* and select()/poll()/etc. functions

- non-blocking I/O setup with fcntl() function
- non-blocking write() fills up buffer as much as possible, then returns
- non-blocking read() returns what’s in buffer, never waits for more

remote procedure calls

goal: I write a bunch of functions

can call them from another machine

some tool + library handles all the details

called *remote procedure calls* (RPCs)

transparency

common hope of distributed systems is *transparency*

transparent = can “see through” system being distributed

for RPC: no difference between remote/local calls

(a nice goal, but…we’ll see)

stubs

typical RPC implementation: generates *stubs*

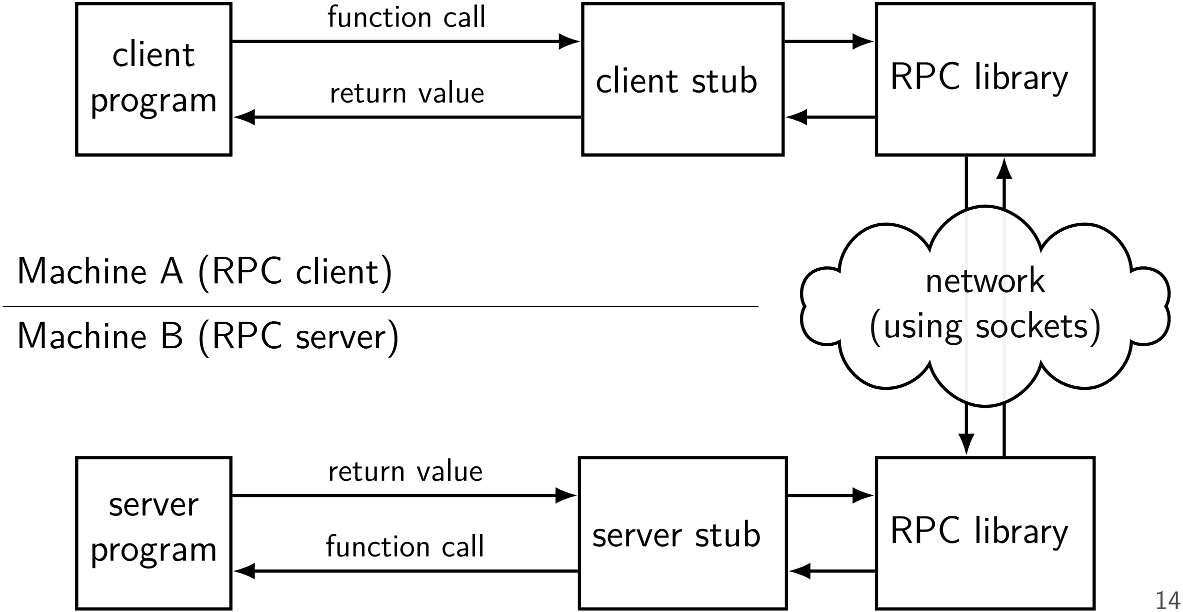
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

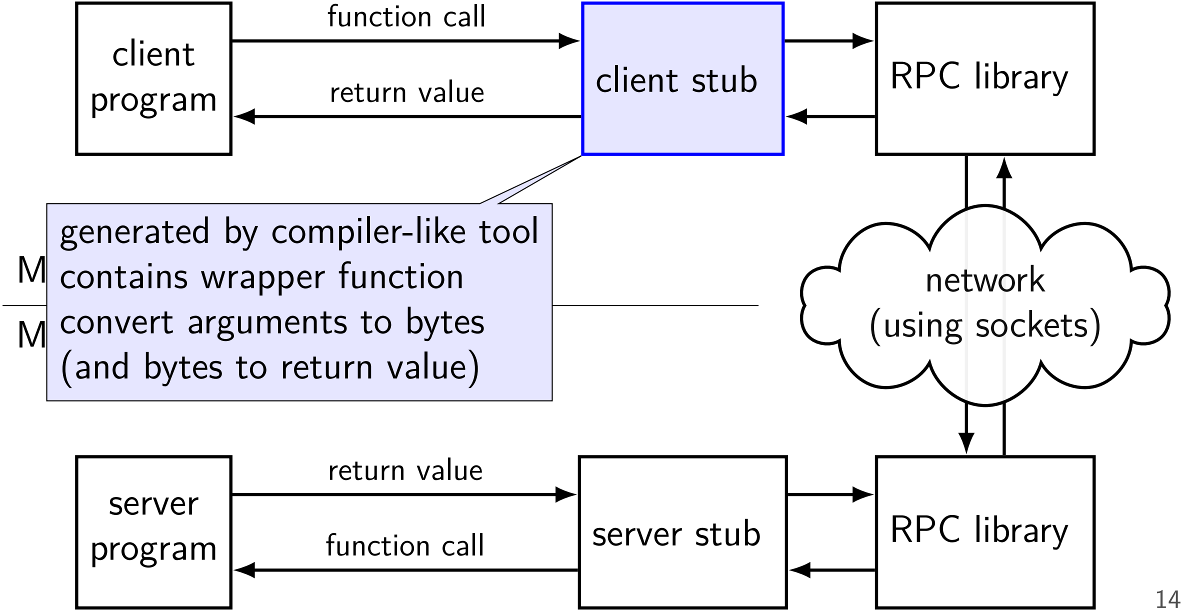
- same prototype are remote procedure

implementing remote procedure? a stub function calls you

typical RPC data flow

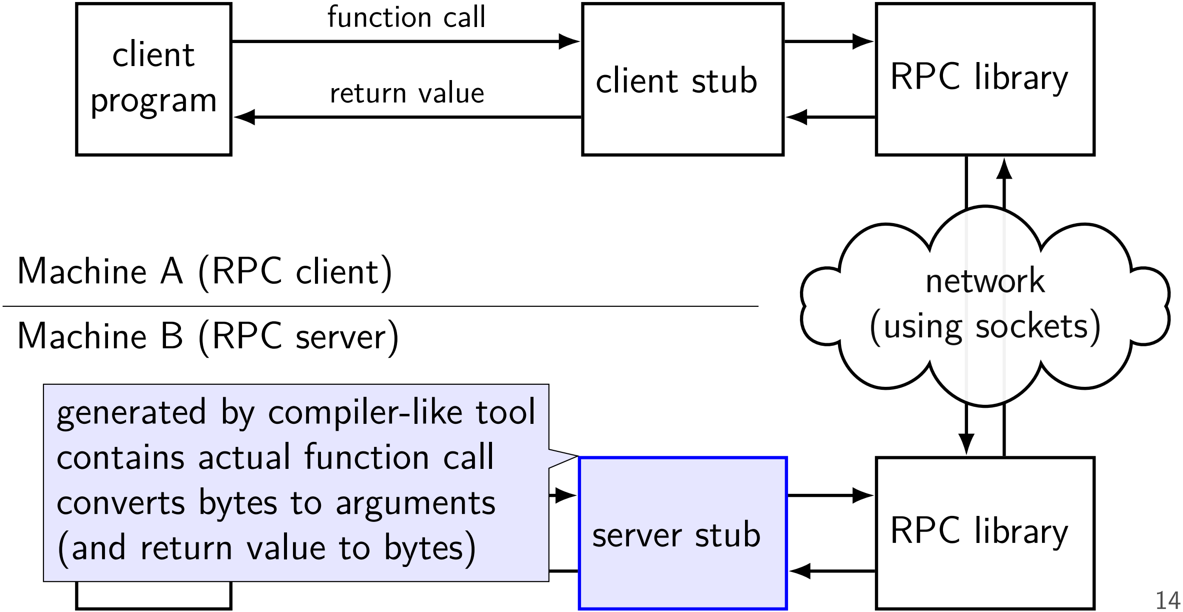


typical RPC data flow



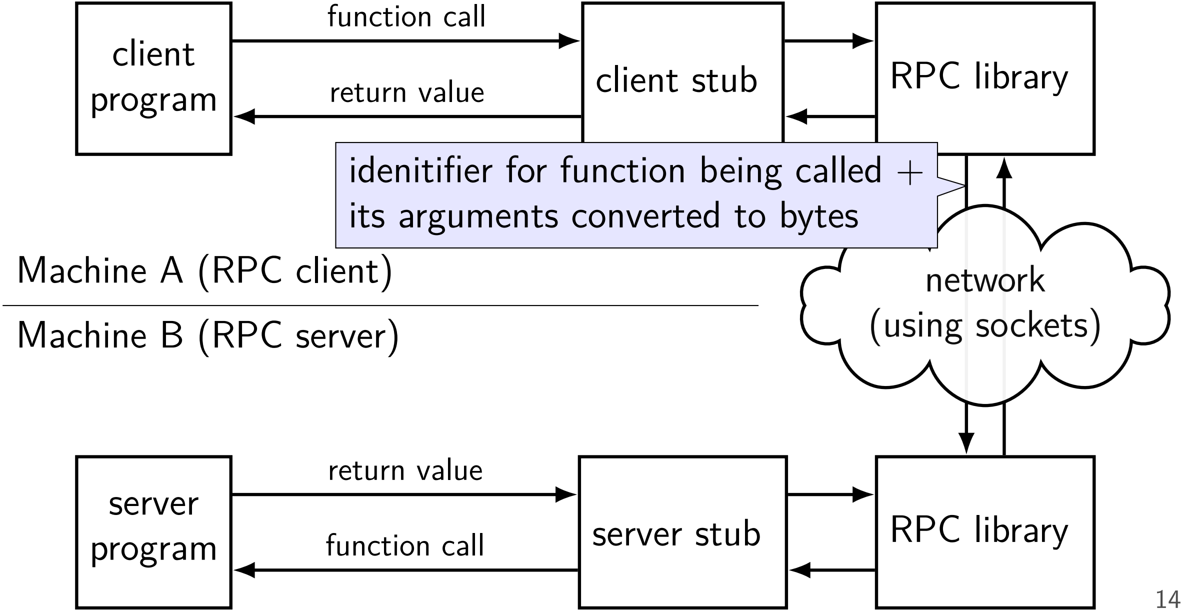
generated by compiler-like tool
contains wrapper function
convert arguments to bytes
(and bytes to return value)

typical RPC data flow

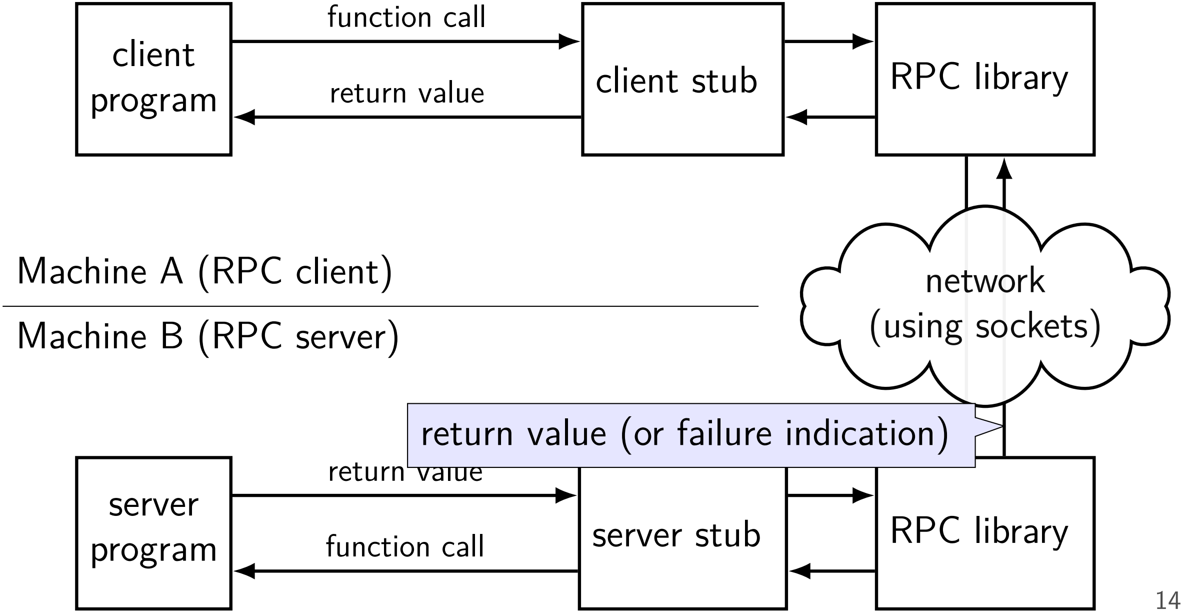


generated by compiler-like tool
contains actual function call
converts bytes to arguments
(and return value to bytes)

typical RPC data flow



typical RPC data flow



return value (or failure indication)

RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server␣name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

---

server:

```
main() {
  dirprotocol_RunServer();
}

// called by server stub
int real_dirprotocol_mkdir(RPCLibraryContext context, char *name) {
   ...
}
```

RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server␣name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

---

server:

```
main() {
  dirprotocol_RunServer();
}

// called by se
int real_dirpro                                    t, char *name) {
  ...
}
```

context to specify and pass info about where the function is actually located

RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server␣name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

---

server:

```
main() {
  dirprotocol_RunServer();
}

// called b
int real_di                                              har *name) {
  ...
}
```

transparency failure:

doesn't look like a normal function call anymore

can we do better than this?

RPC use pseudocode (OO-like)

client:

```
DirProtocol* remote = DirProtocol::connect("server␣name");

// mkdir() is the client stub
result = remote->mkdir("/directory/name");
```

---

server:

```
main() {
    DirProtocol::RunServer(new RealDirProtocol, PORT_NUMBER);
}

class RealDirProtocol : public DirProtocol { public:
  int mkdir(char *name) {
    ...
  }
};
```

marshalling

RPC system needs to send arguments over the network

- and also return values

called *marshalling* or *serialization*

can't just copy the bytes from arguments

- pointers (e.g. `char*`)
- different architectures (32 versus 64-bit; endianness)

interface description langauge

tool/library needs to know:

- what remote procedures exist
- what types they take

typically specified by RPC server author in *interface description language*

- abbreviation: IDL

compiled into stubs and marshalling/unmarshalling code

why IDL? (1)

why don't most tools use the normal source code?

alternate model: just give it a header file

why IDL? (1)

why don't most tools use the normal source code?

alternate model: just give it a header file

missing information (sometimes)

- is `char` array nul-terminated or not?
- where is the size of the array the `int*` points to stored?
- is the `List*` argument being used to modify a list or just read it?
- how should memory be allocated/deallocated?
- how should argument/function name be sent over the network?

why IDL? (2)

why don't most tools use the normal source code?

alternate model: just give it a header file

machine-neutrality and language-neutrality

- common goal: call server from any language, any type of machine
- how big should `long` be?
- how to pass string from C to Python server?

why IDL? (2)

why don't most tools use the normal source code?

alternate model: just give it a header file

machine-neutrality and language-neutrality

- common goal: call server from any language, any type of machine
- how big should `long` be?
- how to pass string from C to Python server?

versioning/compatibility

- what should happen if server has newer/older prototypes than client?

IDL pseudocode + marshalling example

```
protocol dirprotocol {
    1: int32 mkdir(string);
    2: int32 rmdir(string);
}
```

`mkdir("/directory/name")` returning `0`

client sends: `\x01/directory/name\x00`

server sends: `\x00\x00\x00\x00`

GRPC examples

will show examples for gRPC

RPC system originally developed at Google

defines interface description language, message format

uses a protocol on top of HTTP/2

note: gRPC makes some choices other RPC systems don't

GRPC IDL example

```
message MakeDirArgs { required string path = 1; }
message ListDirArgs { required string path = 1; }

message DirectoryEntry {
    required string name = 1;
    optional bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

GRPC IDL example

```
message MakeDirArgs { required string path = 1; }
message ListDirArgs { required string path = 1; }

message DirectoryEntry {
    required string name = 1;
    optional bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

service Directories {
    rpc M
    rpc L                                                ) {}
}
```

messages: turn into C++ classes
with accessors + marshalling/demarshalling functions
part of *protocol buffers* (usable without RPC)

GRPC IDL example

```
message MakeDirArgs { required string path = 1; }
message ListDirArgs { required string path = 1; }

message DirectoryEntry {
    required string name = 1;
    optional bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

service Directories {
    rpc Ma
    rpc Li                                              t) {}
}
```

fields are numbered (can have more than 1 field)
numbers are used in byte-format of messages
allows changing field names, adding new fields, etc.

GRPC IDL example

will become method of C++ class

```
message MakeDirArgs { required string path = 1; }
message ListDirArgs { required string path = 1; }

message DirectoryEntry {
    required string name = 1;
    optional bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

GRPC IDL example

rule: arguments/return value always a *message*

```
message Mak
message ListDirArgs { required string path = 1; }

message DirectoryEntry {
    required string name = 1;
    optional bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->name() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->name() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->name() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->name() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server—>Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
             grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```

RPC non-transparency

setup is not transparent — what server/port/etc.

- ideal: system just knows where to contact?

errors might happen

- what if connection fails?

server and client versions out-of-sync

- can't upgrade at the same time — different machines

performance is very different from local

some gRPC errors

method not implemented

- e.g. server/client versions disagree
- local procedure calls — linker error

deadline exceeded

- no response from server after a while — is it just slow?

connection broken due to network problem

leaking resources?

```
RemoteFile rfh;
stub.RemoteOpen(&context, filename, &rfh);

RemoteWriteRequest remote_write;
remote_write.set_file(rfh);
remote_write.set_data("Some␣text.\n");
stub.RemotePrint(&context, remote_write, ...);
stub.RemoteClose(rfh);
```

---

what happens if client crashes?

does server still have a file open?

related to issue of statefullness

on versioning

normal software: multiple versions of library?

- extra argument for function
- change what function does
- ...

just link against “correct version”

RPC: server gets upgraded out-of-sync with client

want to upgrade functions without breaking old clients

gRPC's versioning

gRPC: messages have field numbers

rules allow adding new *optional* fields

- get message with extra field — ignore it
- (extra field includes field numbers not in *our* source code)
- get message missing optional field — ignore it

otherwise, need to make new methods for each change

- ...and keep the old ones working for a while

versioned protocols

ONC RPC solution: whole service has versions

have implementations of multiple versions in server

verison number is part of every procedures name

RPC performance

local procedure call: ~ 1 ns

system call: ~ 100 ns

network part of remote procedure call

(typical network) $> 400\,000$ ns
(super-fast network) $2\,600$ ns

RPC locally

not uncommon to use RPC on one machine

more convenient alternative to pipes?

allows shared memory implementation

- mmap one common file
- use mutexes+condition variables+etc. inside that memory

network filesystems

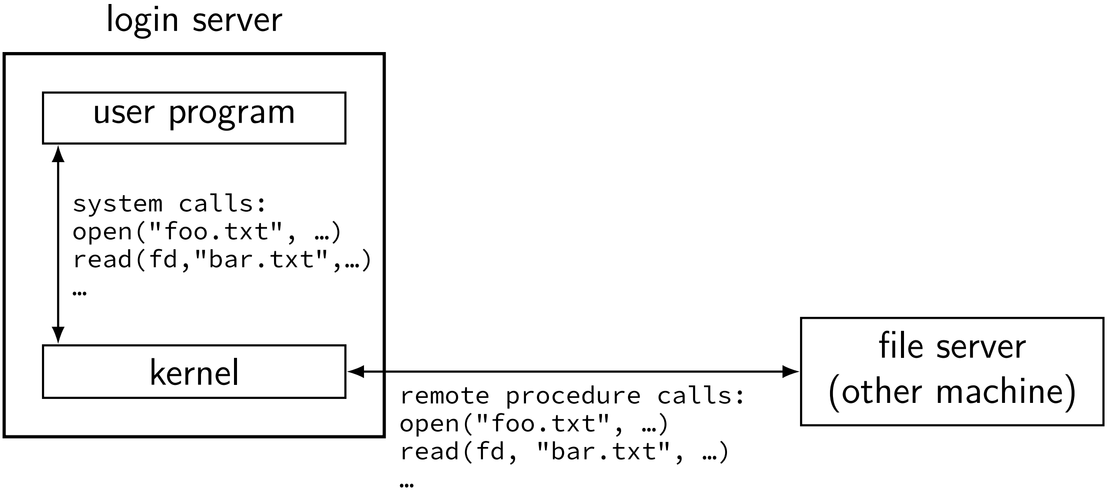
department machines — your files always there

even though several machines to log into

how? there's a *network file server*

filesystem is backed by a remote machine

simple network filesystem



```
system calls:
open("foo.txt", …)
read(fd,"bar.txt",…)
…
```

```
remote procedure calls:
open("foo.txt", …)
read(fd, "bar.txt", …)
…
```

system calls to RPC calls?

just turn system calls into RPC calls?

(or calls to the kernel's internal fileystem abstraction, e.g. Linux's Virtual File System layer)

has some problems:

what state does the server need to store?

what if a client machine crashes?

what if the server crashes?

how fast is this?

state for server to store?

open file descriptors?

- what file
- offset in file

current working directory?

gets pretty expensive across N clients, each with many processes

if a client crashes?

well, it hasn't responded in N minutes, so

can the server delete its open file information yet?

what if its cable is plugged back in and it works again?

if the server crashes?

well, first we restart the server/start a new one...

then, what do clients do?

probably need to restart to?

can we do better?

performance

before: reading/writing files/directories goes to local memory

- lots of work to have use memory to cache, read-ahead

so open/read/write/close/rename/readdir/etc. take microseconds

- open that file? yes, I have the direntry cached
- read from that file? already in my memory

now: they probably take milliseconds+

- open that file? let's ask the server if that's okay
- read from that file? let's copy it from the server

can we do better?

NFSv2

NFS (Network File System) version 2

standardized in RFC 1094 (1989)

based on RPC calls

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

S

file ID: opaque data (support multiple implementations)
example implementation: device+inode number+"generation number"

NFSv2 client versus server

clients: file descriptor →server name, file ID, offset

client machine crashes? mapping automatically deleted

- “fate sharing”

server: convert file IDs to files on disk

- typically find unique number for each file
- usually by inode number

server doesn’t get notified unless client is using the file

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file

- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

file IDs

device + inode + “*generation number*”?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file

- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

Linux filesystems store a “generation number” in the inode

- basically just to help implement things like NFS

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file

"stateless protocol" — no open/close/etc.
each operation stands alone

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset “cookie”) $\rightarrow$
(names and file IDs, next offset “cookie”)

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset “cookie”) → (names and file IDs, next offset “cookie”)

pattern: client storing opaque tokens

for client: remember this, don’t worry about what it means

tokens represent something the server can easily lookup

file IDs: inode, etc.

directory offset cookies: byte offset in directory, etc.

strategy for making stateful service stateless

things NFSv2 didn't do well

performance — each read goes to server?

- would like to cache things in the clients

performance — each write goes to server?

- observation: usually only one user of file at a time
- would like to usually cache writes at clients
- writeback later

offline operation?

- would be nice to work on laptops where wifi sometimes goes out

statefulness

stateful protocol (example: FTP)

- previous things in connection matter
- e.g. logged in user
- e.g. current working directory
- e.g. where to send data connection

stateless protocol (example: HTTP, NFSv2)

- each request stands alone
- servers remember nothing about clients between messages
- e.g. file IDs for each operation instead of file descriptor

stateful versus stateless

in client/server protocols:

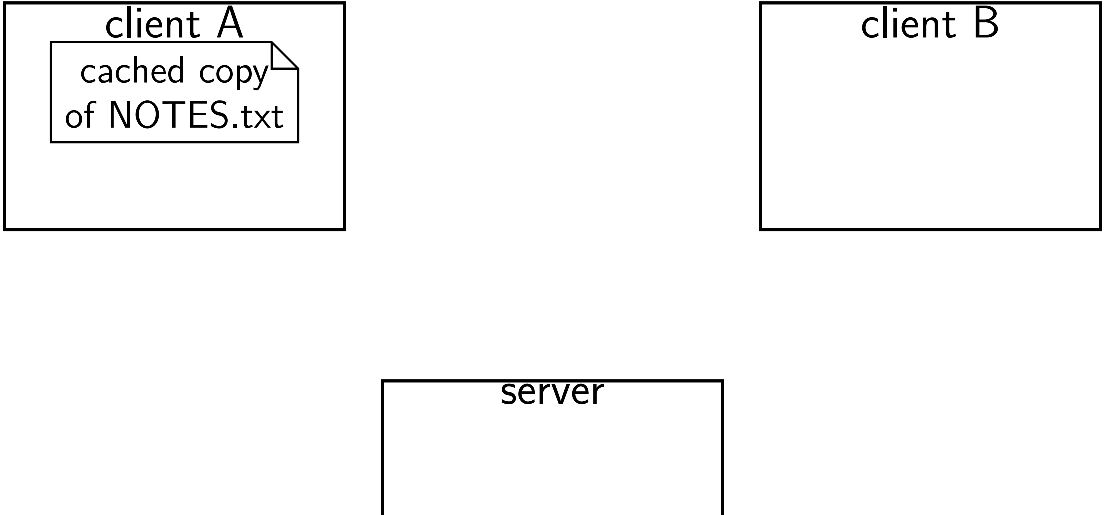
stateless: more work for client, less for server

- client needs to remember/forward any information
- can run multiple copies of server without syncing them
- can reboot server without restoring any client state

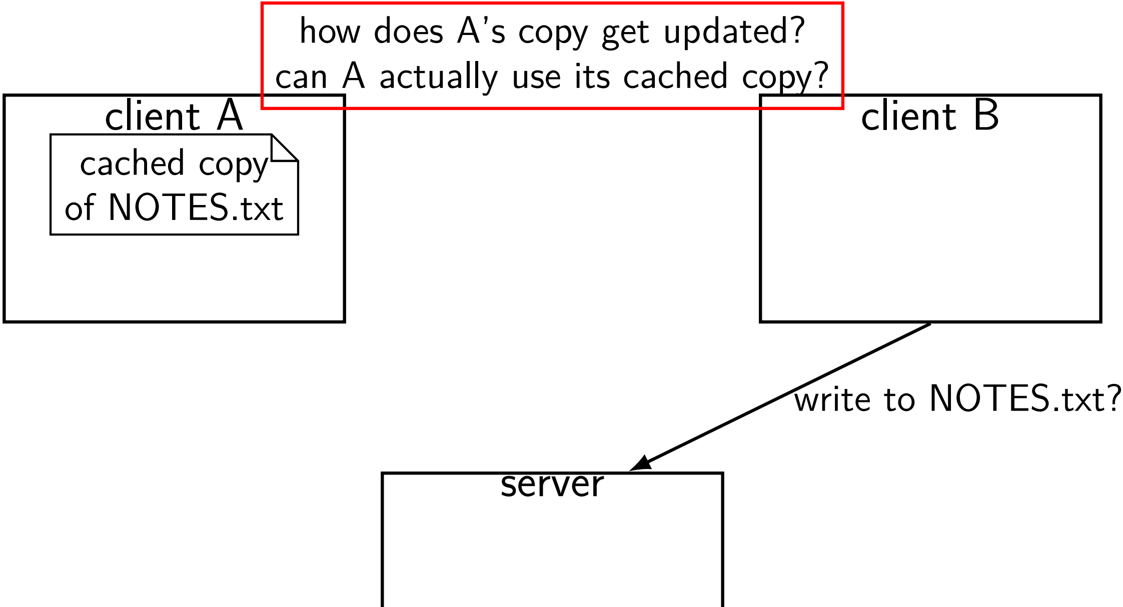
stateful: more work for server, less for client

- client sets things at server, doesn't change anymore
- hard to scale server to many clients (store info for each client
- rebooting server likely to break active connections

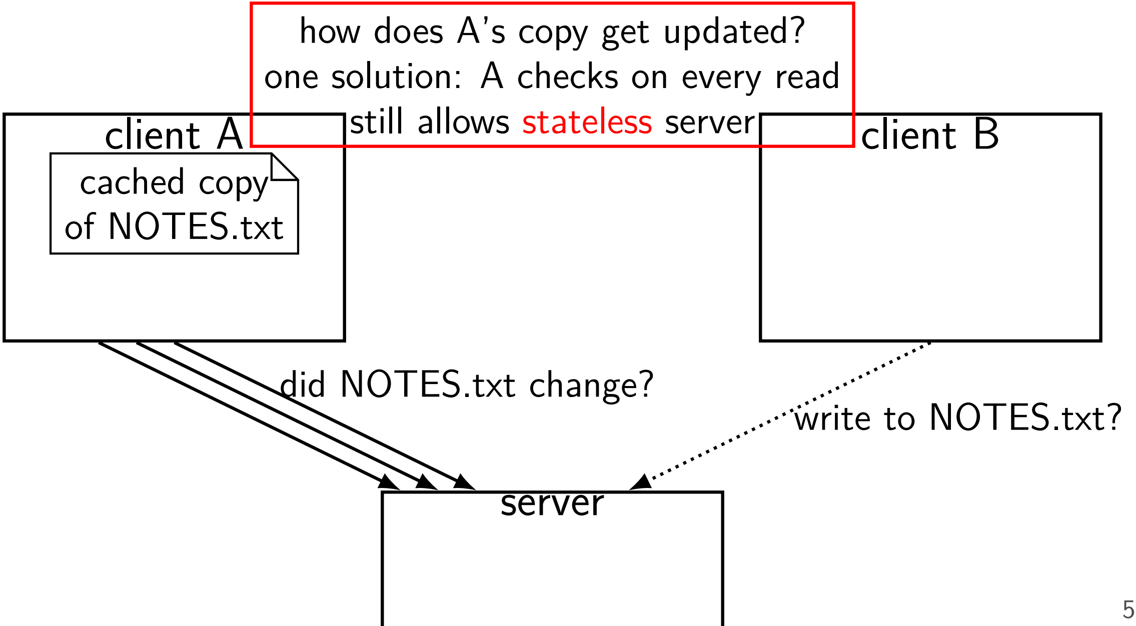
updating cached copies?



updating cached copies?


how does A's copy get updated?
can A actually use its cached copy?
client A
cached copy
of NOTES.txt
client B
write to NOTES.txt?
server

updating cached copies?

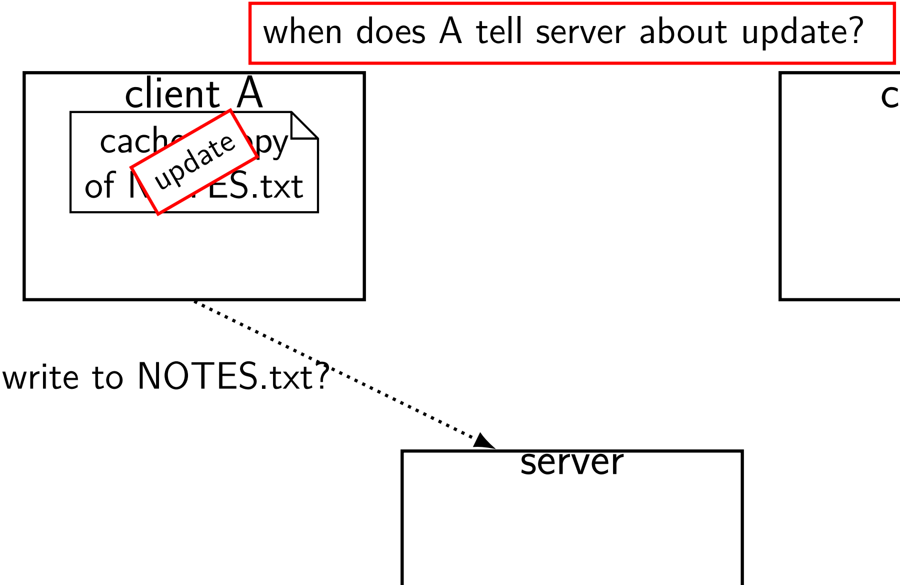


how does A's copy get updated?

one solution: A checks on every read

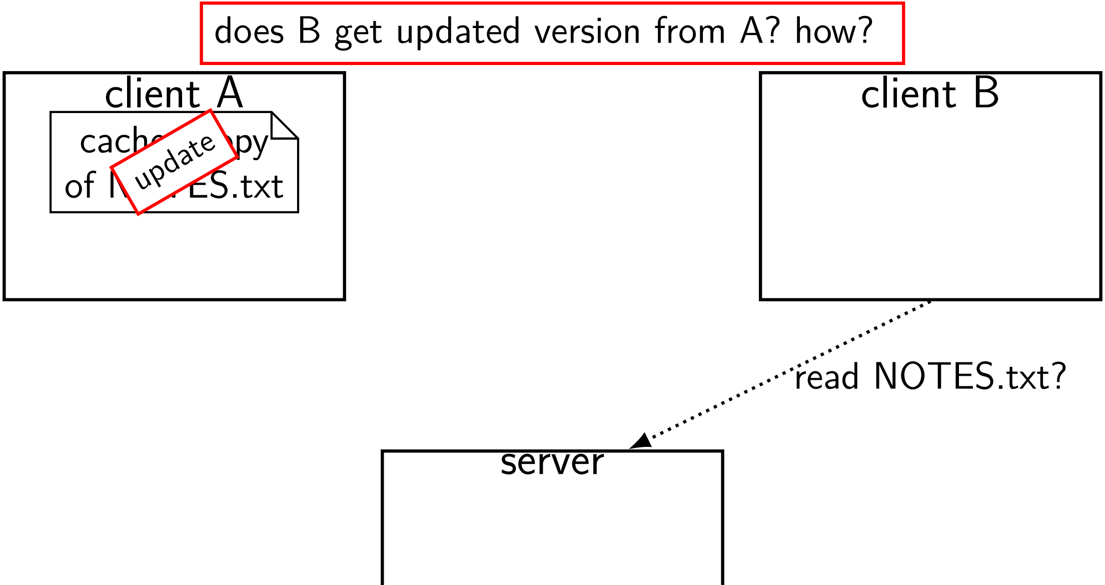
still allows stateless server

updating cached copies?



when does A tell server about update?

updating cached copies?



consistency with stateless server

always check server before using cached version

write through *all* updates to server

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

...but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

…but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

NFSv3's solution: allow inconsistency

typical text editor/word processor

typical word processor:

opening a file:

- open file, read it, load into memory, close it

saving a file:

- open file, write it from memory, close it

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

observation: not things we really expect to work anyways

most applications don't care about accessing file while someone has it open

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

idea: as long as one user loads/saves file at a time, great!

an alternate compromise

application opens a file, read it a day later, result?

- day-old version of file

modification 1: check server/write to server after an amount of time

doesn't need to be much time to be useful

- word processor: typically load/save file in $<$ second

AFSv2

Andrew File System version 2

uses a stateful server

also works file at a time — not parts of file

- i.e. read/write entire files

but still chooses consistency compromise

- still won't support simulatenous read+write from diff. machines well

stateful: avoids repeated 'is my file okay?' queries

NFS versus AFS reading/writing

NFS reading: read/write block at a time

AFS reading: always read/write *entire file*

exercise: pros/cons?

- efficient use of network?
- what kinds of inconsistency happen?
- does it depend on workload?

AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: write whole file

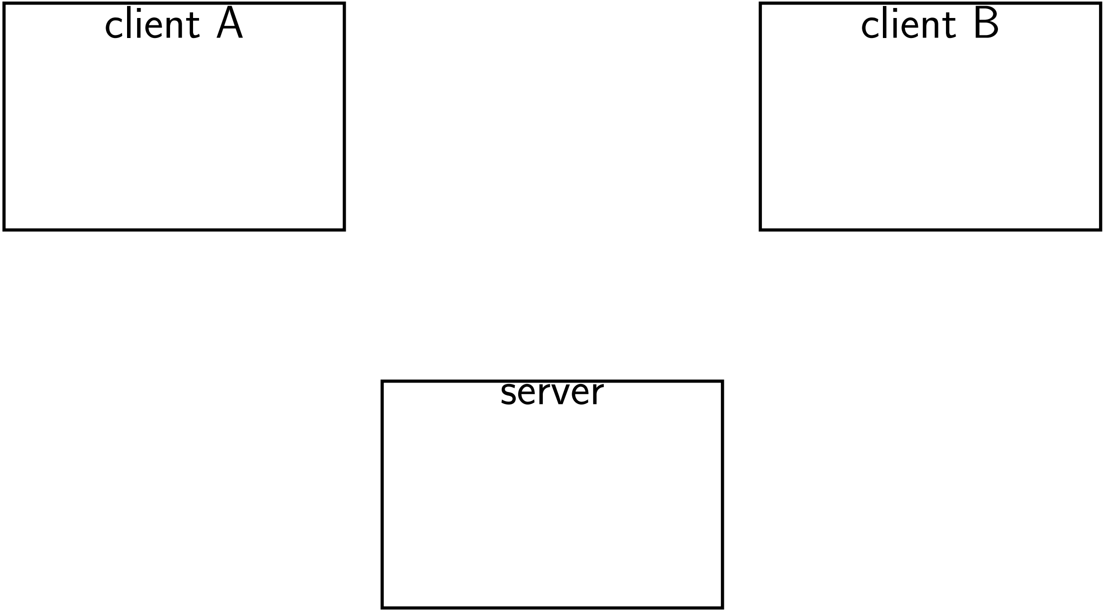
last writer wins

NFS: last writer wins per block

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
NFS: write NOTES.txt block 0	
	close NOTES.txt
	NFS: write NOTES.txt block 0
	NFS: write NOTES.txt block 1
NFS: write NOTES.txt block 1	
NFS: write NOTES.txt block 2	
	NFS: write NOTES.txt block 2

NOTES.txt: 0 from B, 1 from A, 2 from B

AFS caching

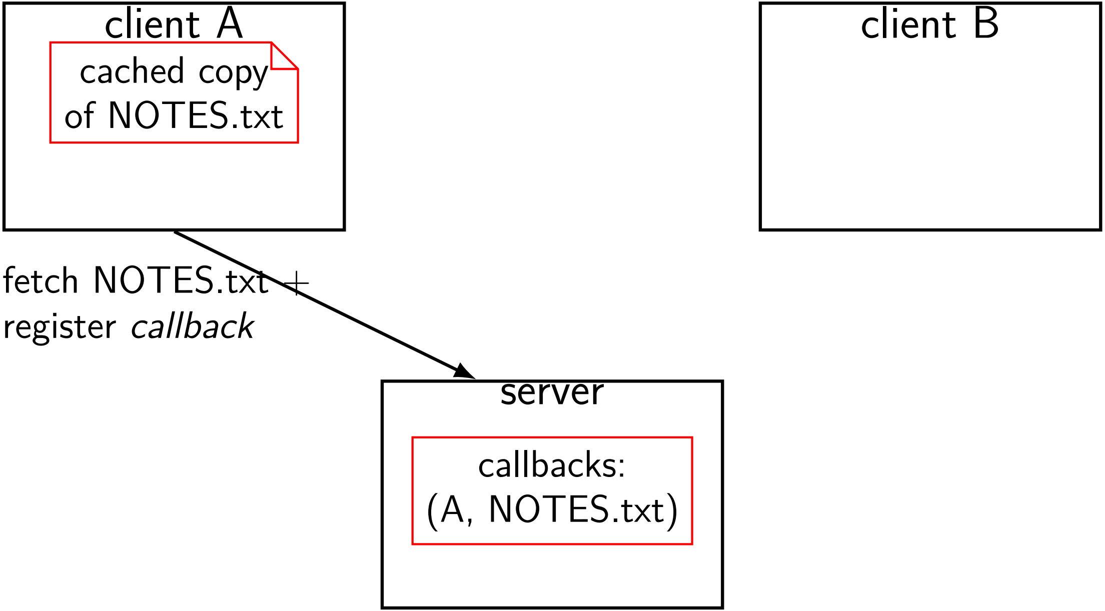


client A

client B

server

AFS caching



client A

cached copy of NOTES.txt

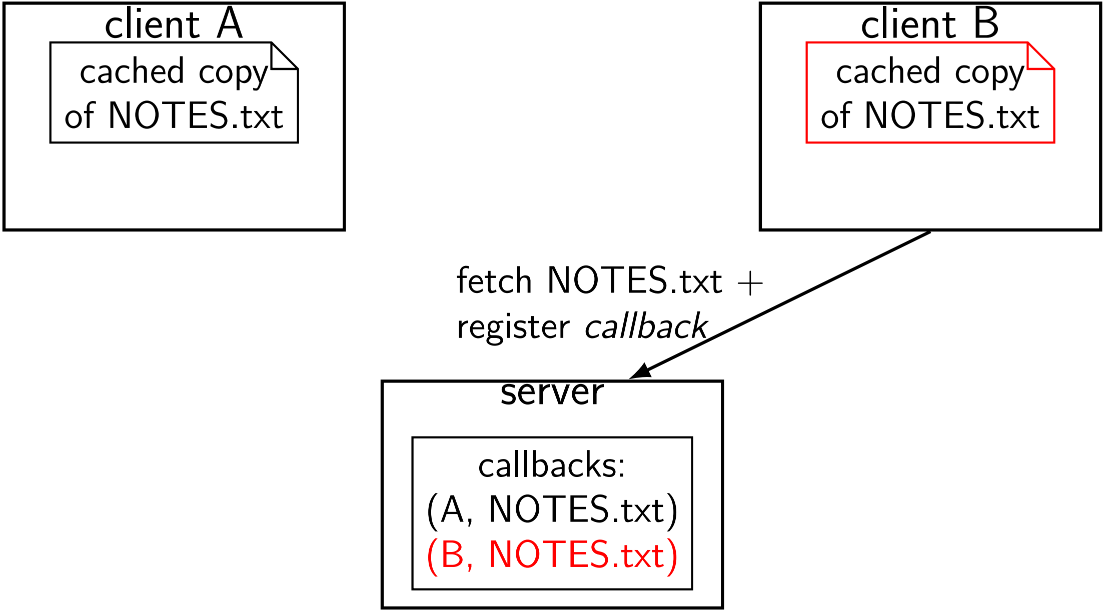
client B

fetch NOTES.txt + register *callback*

server

callbacks: (A, NOTES.txt)

AFS caching



client A

cached copy of NOTES.txt

client B

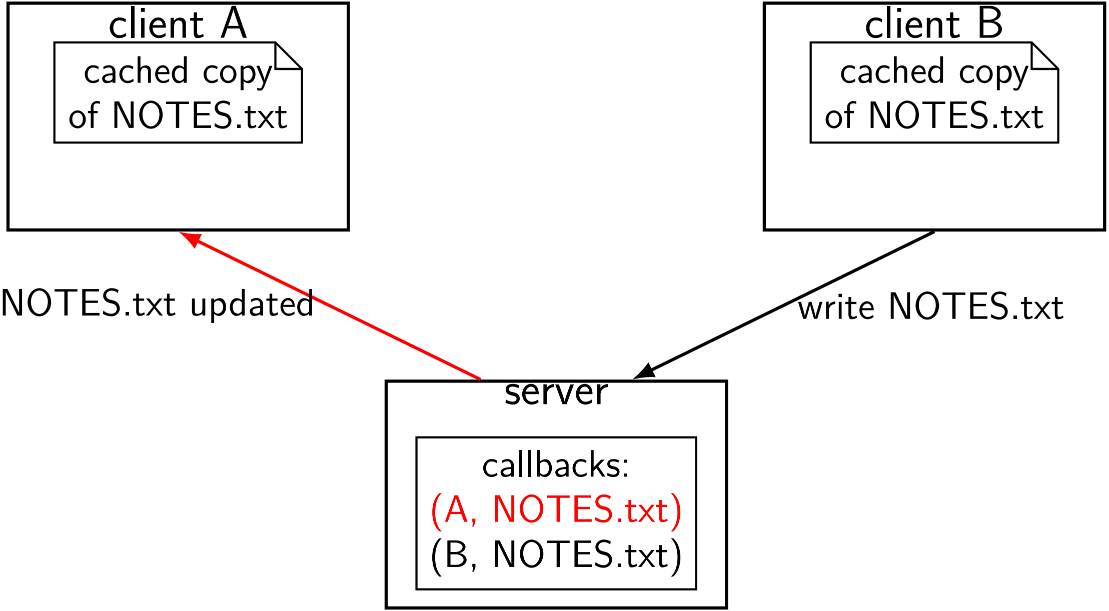
cached copy of NOTES.txt

fetch NOTES.txt + register *callback*

server

callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

AFS caching



client A

cached copy of NOTES.txt

client B

cached copy of NOTES.txt

NOTES.txt updated

write NOTES.txt

server

callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

problem with close-to-open consistency

same issue w/NFS: B can't know about write because server doesn't

(could fix by notifying server earlier)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

close-to-open consistency assumption:
are not accessing file from two places at once

on connections and how they fail

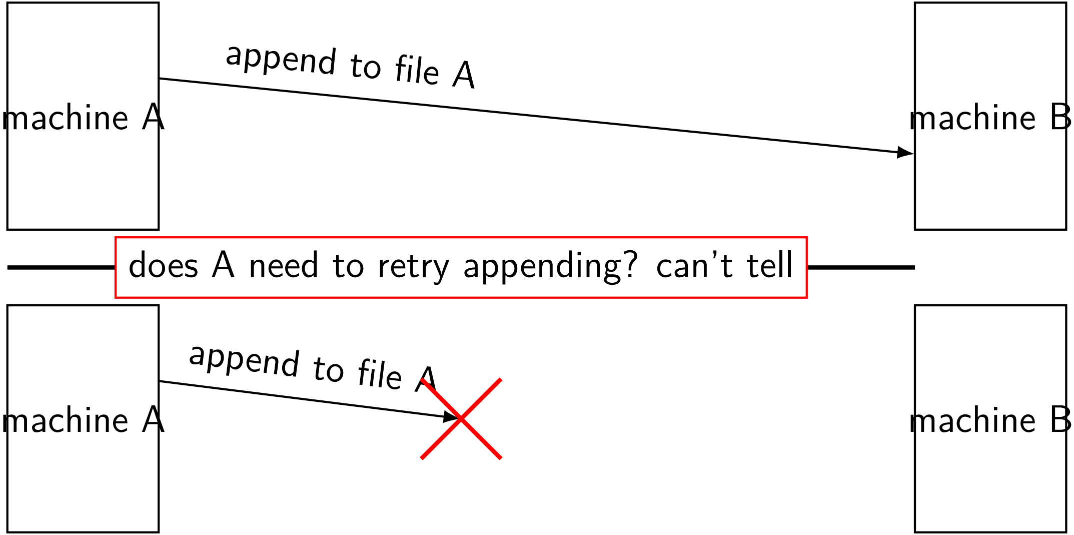
for the most part: don't look at details of connection implementation

...but will do so to explain how things fail

why? important for designing protocols that change things

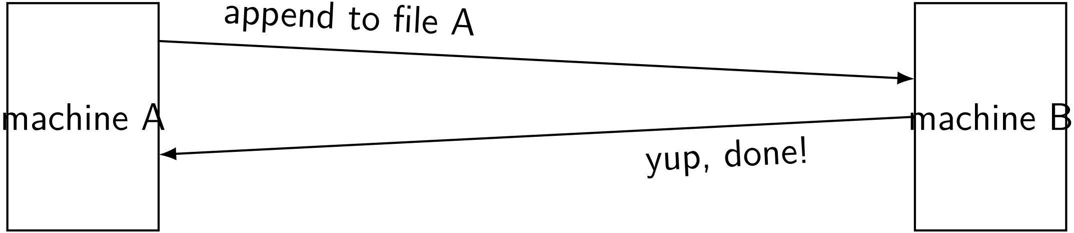
- how do I know if any action took place?

dealing with network failures

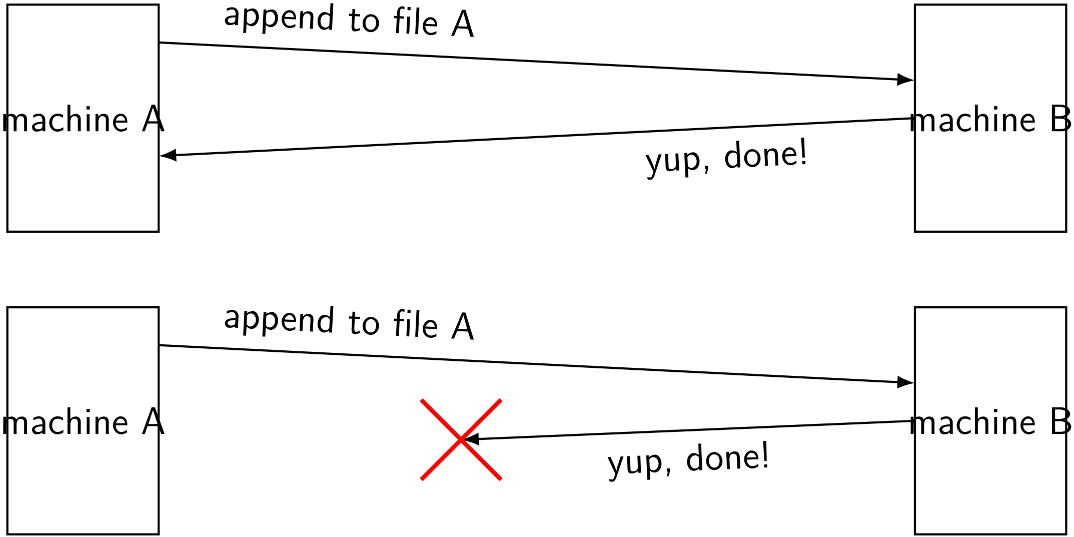


does A need to retry appending? can't tell

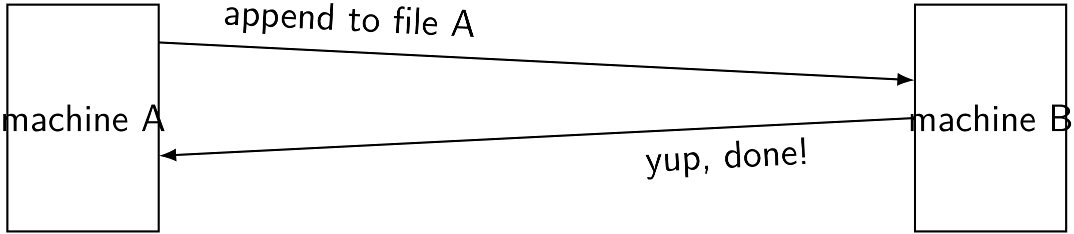
handling failures: try 1



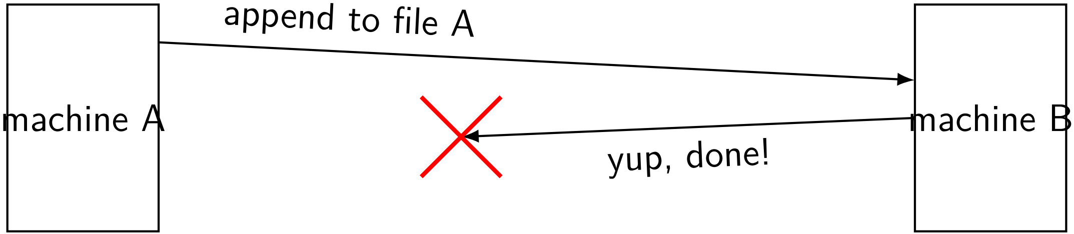
handling failures: try 1



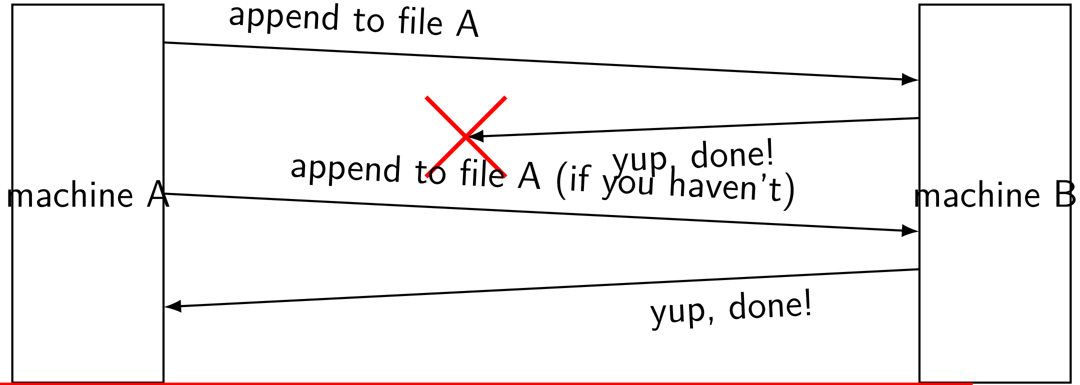
handling failures: try 1



does A need to retry appending? *still* can't tell



handling failures: try 2



retry (in an idempotent way) until we get an acknowledgement
basically the best we can do, but when to give up?

dealing with failures

real connections: acknowledgements + retrying

but have to give up eventually

means on failure — can't always know what happened remotely!

- maybe remote end received data
- maybe it didn't
- maybe it crashed
- maybe it's running, but it's network connection is down
- maybe our network connection is down

also, connection knows *whether program received data*

- not whether program did whatever commands it contained

supporting offline operation

so far: assuming constant contact with server

someone else writes file: we find out

we finish editing file: can tell server right away

good for an office

- my work desktop can almost always talk to server

not so great for mobile cases

- spotty airport/café wifi, no cell reception, …

AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: (over)write whole file

probably losing data!

usually wanted to merge two versions

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

and then…ask user? regenerate file? …?

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

- uses version IDs to decide what to update

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

- uses version IDs to decide what to update

DropBox, etc. probably similar idea?

version ID?

not a version number?

actually a *version vector*

version number for each machine that modified file

- number for each server, client

allows use of multiple servers

- if servers get desync'd, use version vector to detect
- then do, uh, something to fix any conflicting writes

file locking

so, your program doesn't like conflicting writes

what can you do?

if offline operation, probably not much...

otherwise file locking

except it often doesn't work on NFS, etc.

advisory file locking with fcntl

```c
int fd = open(...);
struct flock lock_info = {
    .l_type = F_WRLCK,  // write lock; RDLOCK also available
    // range of bytes to lock:
    .l_whence = SEEK_SET, l_start = 0, l_len = ...
};
/* set lock, waiting if needed */
int rv = fcntl(fd, F_SETLKW, &lock_info);
if (rv == −1) { /* handle error */ }
/* now have a lock on the file */

/* unlock --- could also close() */
lock_info.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock_info);
```

advisory locks

fcntl is an *advisory* lock

doesn't stop others from accessing the file...

unless they always try to get a lock first

POSIX file locks are horrible

actually two locking APIs: fcntl() and flock()

fcntl: *not* inherited by fork

fcntl: closing any fd for file release lock

even if you dup2'd it!

fcntl: maybe sometimes works over NFS?

flock: less likely to work over NFS, etc.

fcntl and NFS

seems to require extra state at the server

typical implementation: separate *lock server*

not a stateless protocol

lockfiles

use a separate *lockfile* instead of "real" locks

- e.g. convention: use `NOTES.txt.lock` as lock file

lock: create a *lockfile* with `link()` or `open()` with `O_EXCL`

- can't lock: link()/open() will fail "file already exists"
- for current NFSv3: should be single RPC calls that always contact server
- some (old, I hope?) systems: link() atomic, open() O_EXCL not

unlock: remove the lockfile

- annoyance: what if program crashes, file not removed?