RPC / failure

last time

redo logging (finish)
- (weird?) choice not to use redo logging for everything

client/server → peer-to-peer

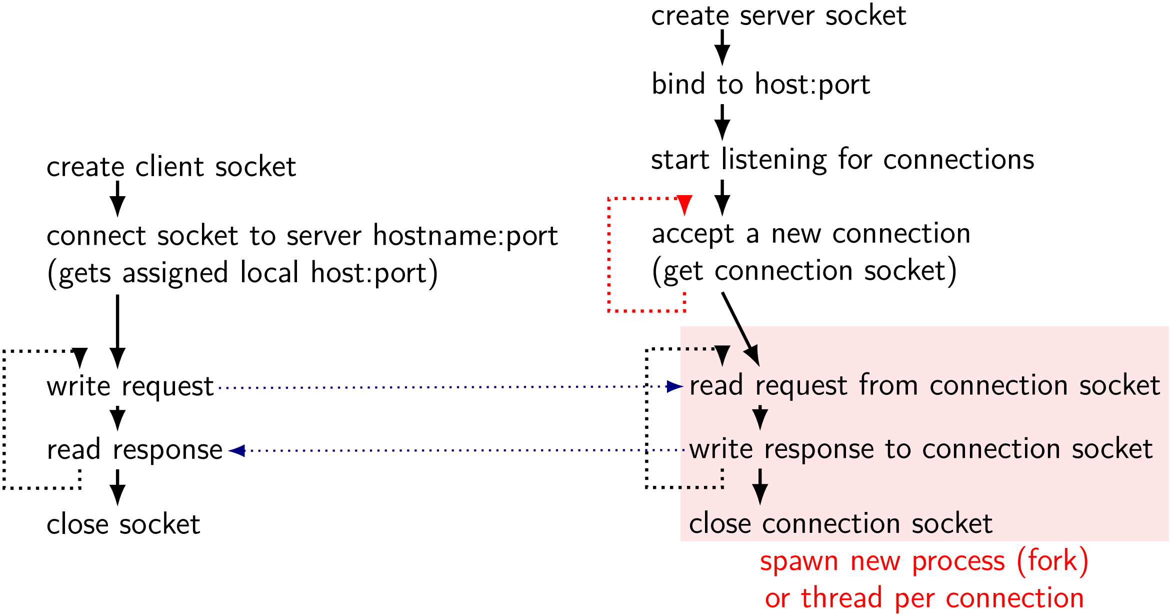
reasons to use distributed systems

mailbox and connnection models

names versus addresses
- domain name system — distributed, hierarchical database
- port numbers

sockets: connections as file descriptors
- bind: set local address
- accept: get connection (as *new* file descriptor)
- connect: make current file descriptor connection to server

client/server flow (multiple connections)



create server socket

bind to host:port

start listening for connections

accept a new connection
(get connection socket)

create client socket

connect socket to server hostname:port
(gets assigned local host:port)

write request

read response

close socket

read request from connection socket

write response to connection socket

close connection socket

spawn new process (fork)
or thread per connection

sockets: missing pieces

translating names to IP address + port number — `getaddrinfo`

- construct arguments for bind (set local address) + connect (set remote address)
- handles using DNS and both IPv4 and IPv6

local/Unix domain sockets

POSIX defines sockets that only work on local machine

example use: apps talking to display manager program

- want to display window? connect to special socket file
- probably don't want this to happen from remote machines

equivalent of name+port: socket file

- appears as a special file on disk

we will use this in assignment

- but you won't directly write code that uses POSIX API

Unix-domain sockets: client example

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, "/path/to/server.socket");
int fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (connect(fd, &server_addr, sizeof(server_addr)) < 0)
    handleError();
... // use 'fd' here
```

Unix-domain sockets: client example

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, "/path/to/server.socket");
int fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (connect(fd, &server_addr, sizeof(server_addr)) < 0)
    handleError();
... // use 'fd' here
```

Unix-domain sockets on my laptop

```
cr4bd@reiss–lenovo:~$ netstat ––unix  –a
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I–Node   Path
unix  2      [ ]         DGRAM                    40077    /run/user/1000/syst
unix  2      [ ACC ]     SEQPACKET  LISTENING     844      /run/udev/control
unix  2      [ ACC ]     STREAM     LISTENING     40080    /run/user/1000/syst
unix  2      [ ACC ]     STREAM     LISTENING     40084    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37867    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37868    /run/user/1000/bus
unix  2      [ ACC ]     STREAM     LISTENING     37869    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37870    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     60556115 /var/run/cups/cups.
unix  2      [ ACC ]     STREAM     LISTENING     37871    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37874    /run/user/1000/keyr
unix  2      [ ACC ]     STREAM     LISTENING     49772163 /run/user/1000/puls
unix  2      [ ACC ]     STREAM     LISTENING     49772158 /run/user/1000/puls
unix  2      [ ACC ]     STREAM     LISTENING     59062776 /run/user/1000/spee
unix  2      [ ACC ]     STREAM     LISTENING     32980    @/tmp/.X11–unix/X0
unix  2      [ ACC ]     STREAM     LISTENING     60557382 /run/cups/cups.sock
...
```

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*

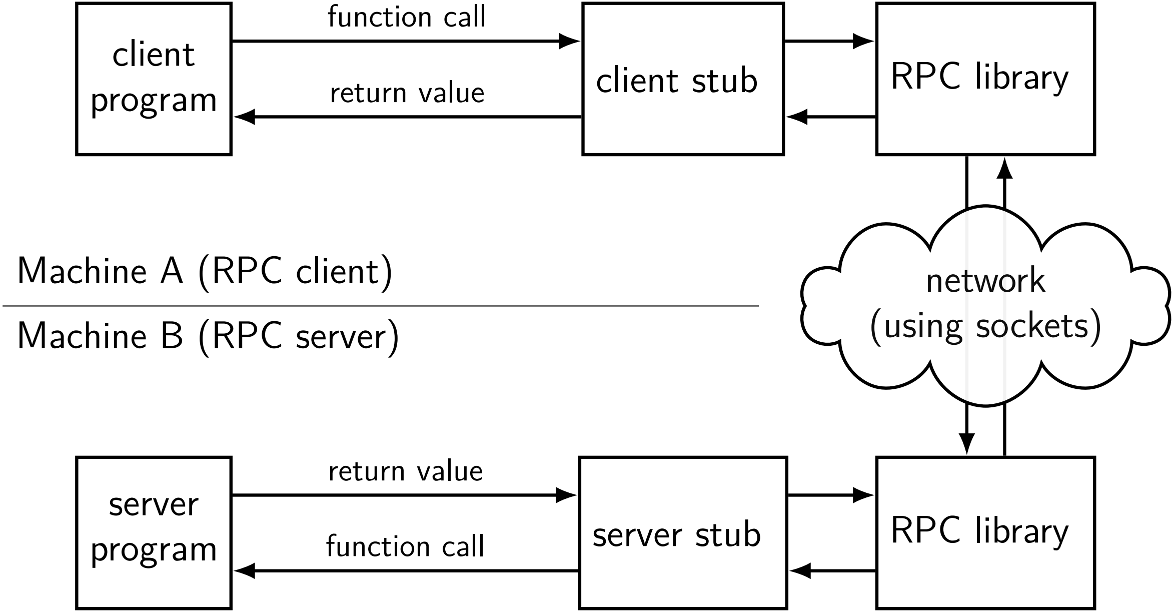
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

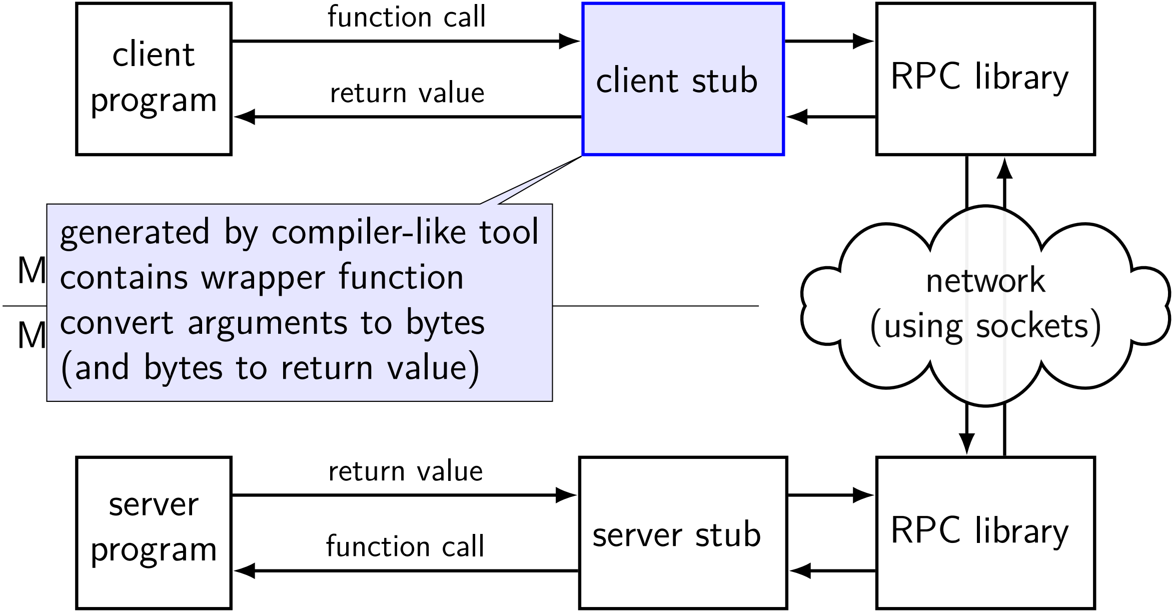
- same prototype are remote procedure

implementing remote procedure? a stub function calls you

typical RPC data flow

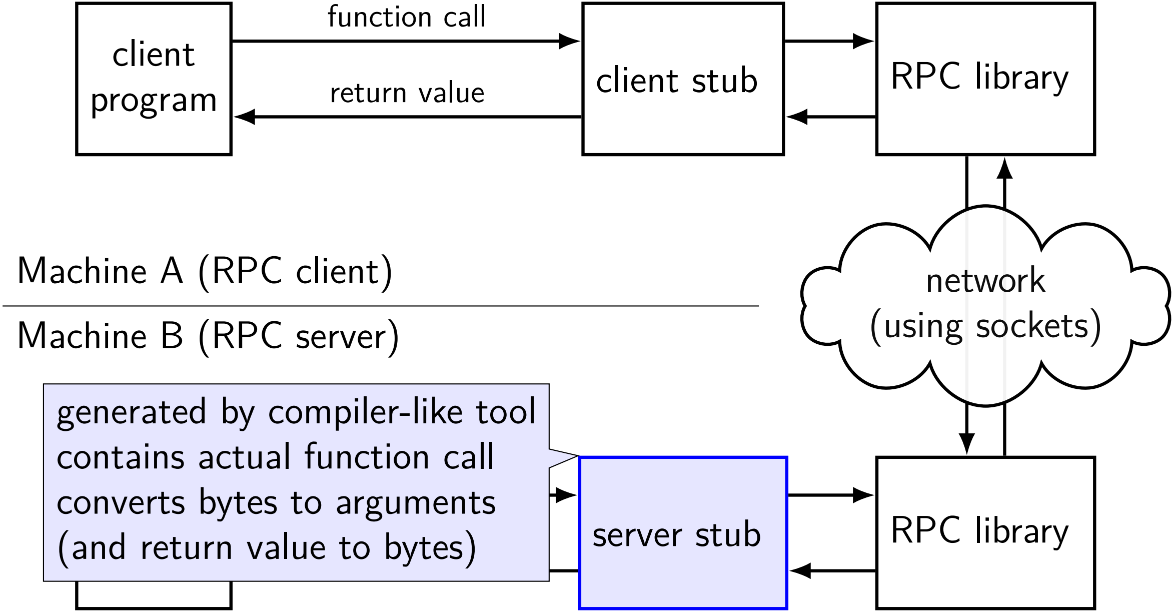


typical RPC data flow



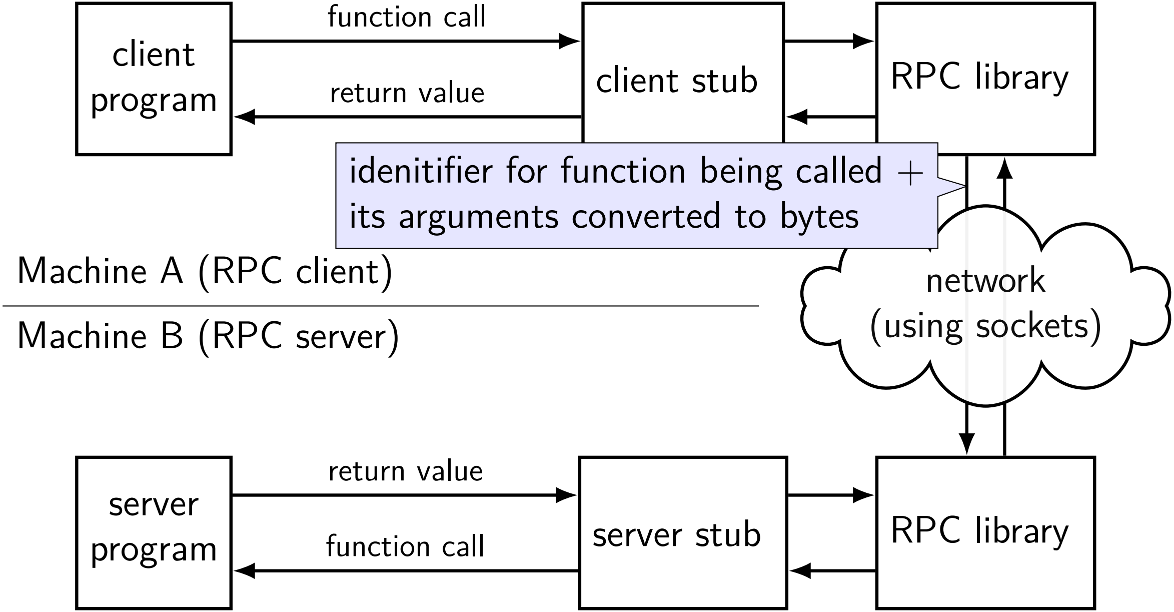
generated by compiler-like tool
contains wrapper function
convert arguments to bytes
(and bytes to return value)

typical RPC data flow

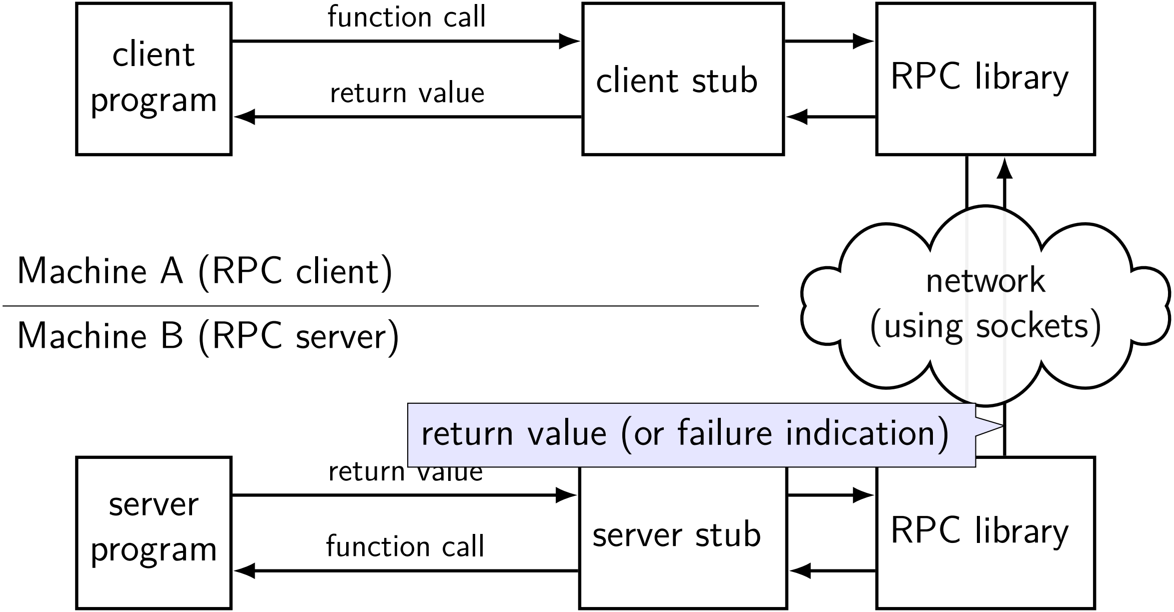


generated by compiler-like tool
contains actual function call
converts bytes to arguments
(and return value to bytes)

typical RPC data flow



typical RPC data flow



return value (or failure indication)

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    exce
      co

    retu
```

client: calls "MakeDirectory" function on server

local-only code would have been:

`MakeDirectory(path="/directory/name")`

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDir                                       ory/name"))
except:
  # handle error
```

server: defines “MakeDirectory” function

local-only code would have been:

```
def MakeDirectory(path):
    ...
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except
      conte
                                                        )
    return
```

stub and *context* to pass info about where the function is actually located (on client) and how it was called (on server)

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except (
      conte                                              )

    return
```

gRPC requires exactly one arguments object
to simplify library/cross-language compatability
some other RPC systems are more flexible

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    exc
      c

    ret
```

generated code (“server stub”) defines base class
server subclass overrides methods to provide remote calls
so it’s easy for library to find them

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return Empty()
```

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?

gRPC IDL example + marshalling

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

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

example possible format (*not what gRPC actually does*):

MakeDirectory(MakeDirArgs(path="/foo")) becomes:

```
\x0dMakeDirectory\x01\x04/foo
```

`0x0d` = length of 'MakeDirectory'
`0x04` = length of '/foo'

GRPC examples

will show examples for gRPC

RPC system originally developed at Google

what we'll use for upcoming assignment

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

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

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

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

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

GRPC IDL example

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

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

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

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

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

GRPC IDL example

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

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

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

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

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

```
syntax="proto3";
message MakeDirA
message ListDirArgs { string path = 1; }

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

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

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

will become method of Python class

GRPC IDL example

rule: arguments/return value always a *message*

```
syntax="pro
message Mak
message ListDirArgs { string path = 1; }

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

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

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

RPC server implementation (method 1)

```
import dirproto_pb2
import dirproto_pb2_grpc

class DirectoriesImpl(dirproto_pb2_grpc.DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    print("MakeDirectory␣called␣with␣path=", request.path)
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return dirproto_pb2.Empty()
```

RPC server implementation (method 2)

```
import dirproto_pb2, dirproto_pb2_grpc
from dirproto_pb2 import DirectoryList, DirectoryEntry

class DirectoriesImpl(dirproto_pb2_grpc.DirectoriesServicer):
  ...
  def ListDirectory(self, request, context):
    try:
      result = DirectoryList()
      for file_name in os.listdir(request.path)
        result.entries.append(DirectoryEntry(name=file_name, ...))
    except OSError as err:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS␣returned␣error:␣{}".format(err))
    return result
```

RPC server implementation (starting)

```
# create server that uses thread pool with
# three threads to run procedure calls
server = grpc.server(
    futures.ThreadPoolExecutor(max_workers=3)
)
# DirectoriesImpl() creates instance of implementaiton class
# add_DirectoryServicer_to_server part of generated code
dirproto_pb2_grpc.add_DirectoryServicer_to_server(
    DirectoriesImpl()
)
server.add_insecure_port('127.0.0.1:12345')
server.start()  # runs server in separate thread
```

RPC client implementation (method 1)

```
from dirproto_pb2_grpc import DirectoriesStub
from dirproto_pb2 import MakeDirectoryArgs

channel = grpc.insecure_channel('127.0.0.1:43534')
stub = DirectoriesStub(channel)
args = MakeDirectoryArgs(path="/directory/name")
try:
  stub.MakeDirectory(args)
except grpc.RpcError as error:
  ... # handle error
```

RPC client implementation (method 2)

```
from dirproto_pb2_grpc import DirectoriesStub
from dirproto_pb2 import ListDirectoryArgs

channel = grpc.insecure_channel('127.0.0.1:43534')
stub = DirectoriesStub(channel)
args = ListDirectoryArgs(path="/directory/name")
try:
  result = stub.ListDirectory(args)
  for entry in result.entries:
    print(entry.name)
except grpc.RpcError as error:
  ... # handle error
```

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

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

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

network failures: two kinds

messages lost

messages delayed/reordered

network failures: message lost?

detect with acknowledgements (“yes I got it”)

can recover by retrying

can’t distinguish: original message lost or acknowledgment lost

can’t distinguish: machine crashed or network down/slow for a while

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

exercise: RPC failure scenarios

RPC with MakeDirectory(”foo”)

option A: client stub returns when sent to server

option B: client stub waits for server to return OK

for now, assume only *network* failures

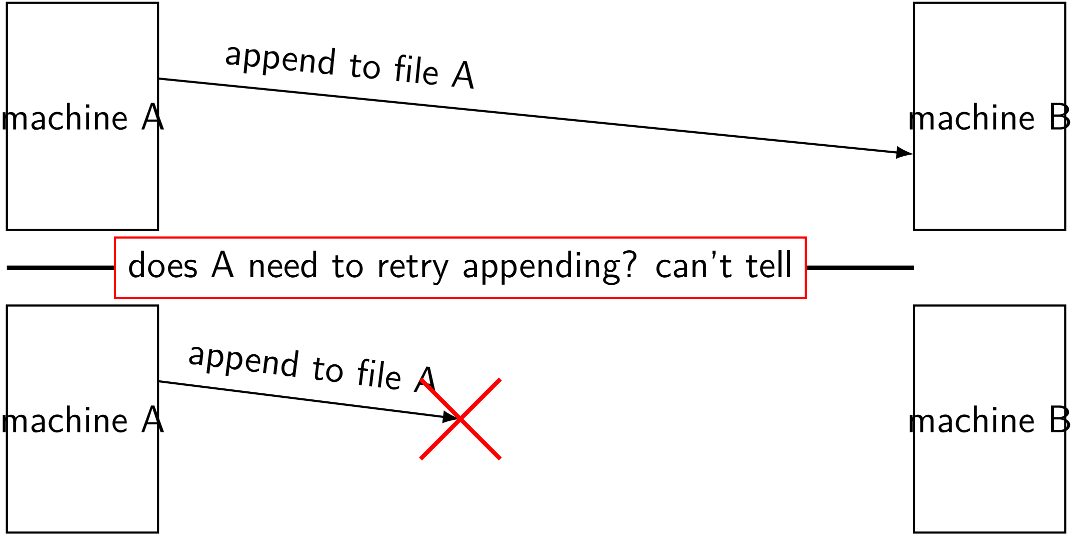
I call MakeDirectory(”foo”) and it throws an exception:

with Option A: could directory have been created?
with Option B: could directory have been created?

I call MakeDirectory(”foo”) and it throws no exception:

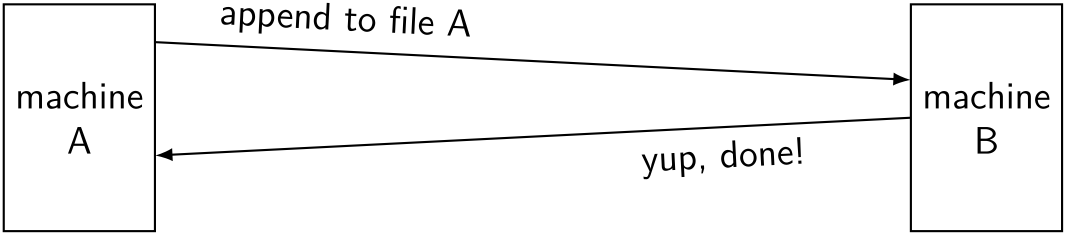
with Option A: could directory have NOT been created?
with Option B: could directory have NOT been created?

dealing with network message lost

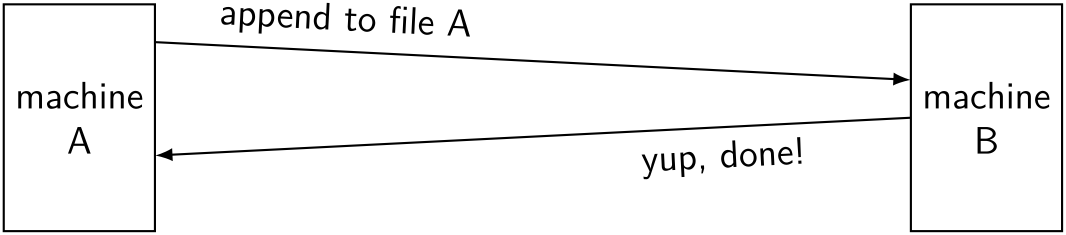


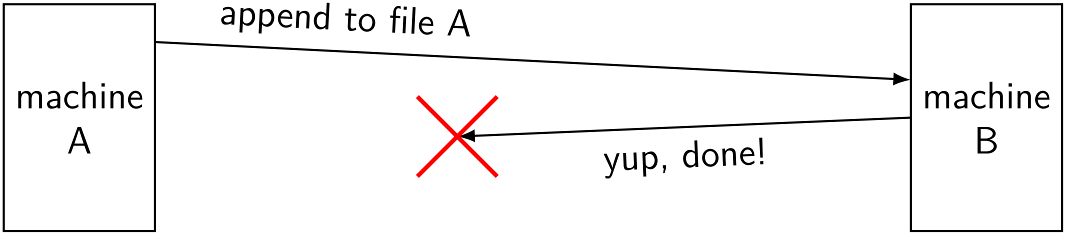
does A need to retry appending? can't tell

handling failures: try 1

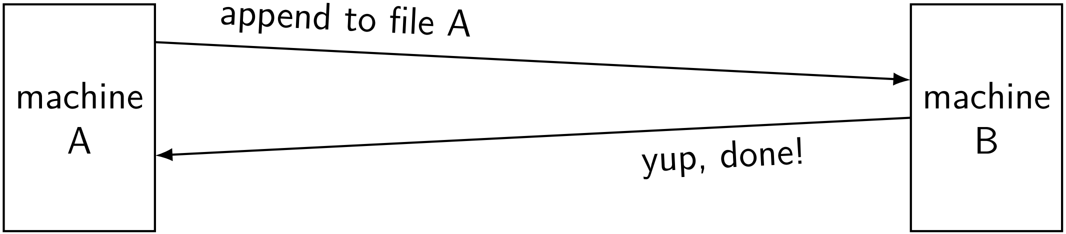


handling failures: try 1

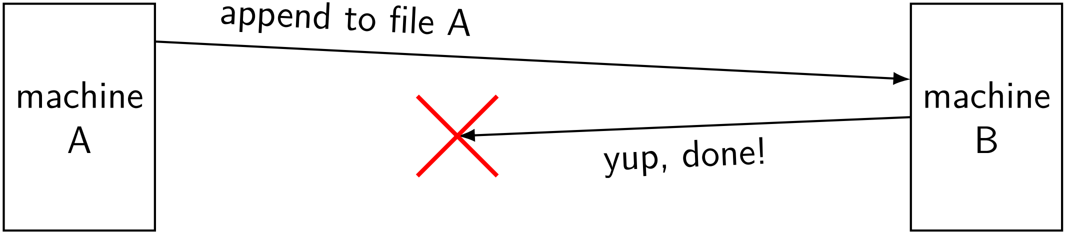




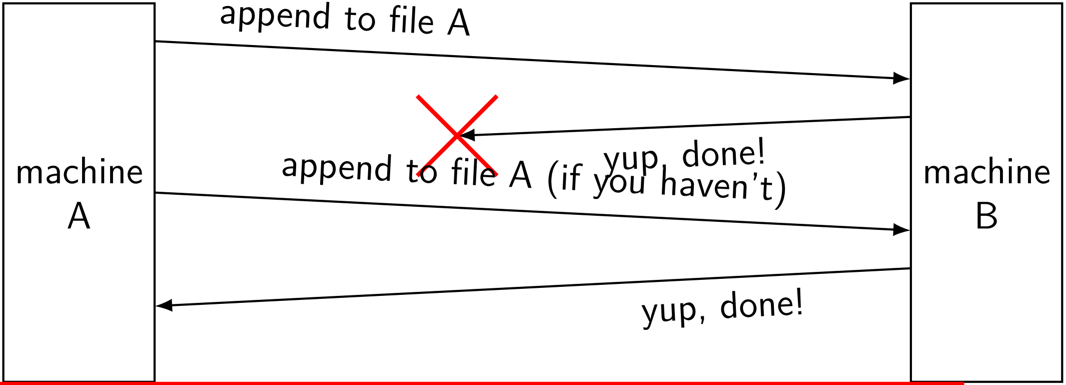
handling failures: try 1



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?

network failures: message reordered?

can detect with sequence numbers

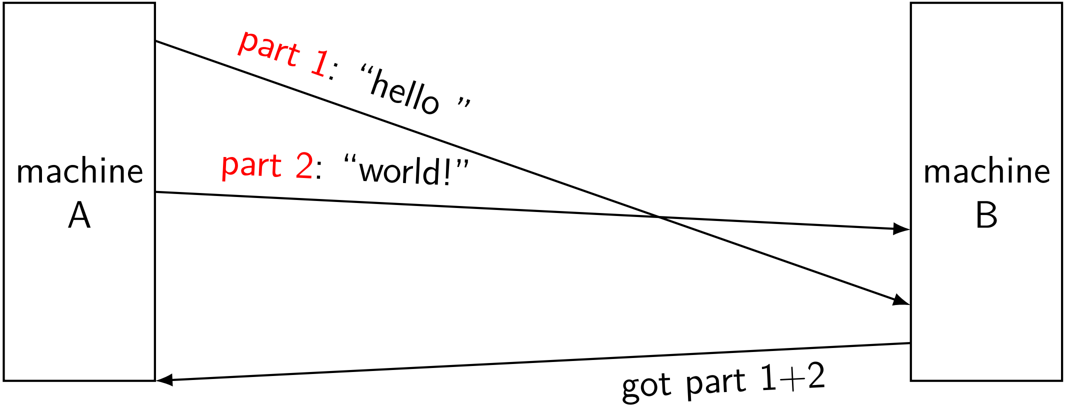
connection protocols do this

RPC abstraction — generally doesn't

- potentially receive 'stale' RPC call

can't distinguish: message lost or just delayed and not received yet

handling reordering



failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

two models of machine failure

fail-stop

failing machines stop responding/don't get messages

or one always detects they're broken and can ignore them

Byzantine failures

failing machines do the worst possible thing

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F + 1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F+1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

distributed transaction problem

distributed transaction

two machines both agree to do something *or not do something*

even if *a machine fails*

primary goal: *consistent* state

secondary goal: do it if nothing breaks

distributed transaction example

course database across many machines

machine A and B: student records

machine C: course records

want to make sure machines agree to add students to course

no confusion about student is in course even if failures

“consistency”

okay to say “no” — if possible, can retry later

naive distributed transaction? (1)

machine A and B: student records; machine C: course records

any machine can be queried directly for info (e.g. by SIS web interface)

proposed add student to course procedure:

execute code on A or B where student is stored

tell C: add student to course

wait for response from C (if course full, return error)

locally: add student to course

exericse (1)

seperate student (local) + course (remote) records

tell remote: add student to course

then locally: add student to course

if no failures, which are possible to observe from third machine (that asks student/course machines for current records)?

- A student record: in course; course record: not in course; but if double checking: both agree
- B same as A, but if double-checking both *do not* agree
- C student record: not in course; course record: in course; but if double checking: both agree
- D same as C, but if double-checking both *do not* agree

exericse (2)

seperate student (local) + course (remote) records

tell remote: add student to course

then locally: add student to course

if failures, which are possible to observe from third machine (that asks student/course machines for current records)?

- A student record: in course; course record: not in course; but if double checking: both agree
- B same as A, but if double-checking both *do not* agree
- C student record: not in course; course record: in course; but if double checking: both agree
- D same as C, but if double-checking both *do not* agree

backup slides

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

renaming fields? doesn't matter, just number changes

rules allow adding new (optional) fields

- get message with extra field — ignore it
- get message missing field — default/null value

otherwise, need to make new methods for each change

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

versioned protocols

alternative approach: version numbers in protocol/messages

server can implement multiple versions

eventually discard old versions:

gRPC: returning errors

any RPC can result in an error

- both errors from libraries and from RPCs can use same API

Python client: throws a grpc.RpcError exception

- no support for custom exceptions types (probably because tricky to make language-neutral)

C++ client: method return value is a Status object

- result of method ‘returned’ by modifying result object passed via pointer
- (for historical reasons, Google doesn’t like C++ exceptions)

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?

```
stub = ...
remote_file_handle = stub.RemoteOpen(filename)
write_request = RemoteWriteRequest(
    file_handle=remote_file_handle,
    data="Some␣text.\n"
)
stub.RemotePrint(write_request)
stub.RemoteClose(remote_file_handle)
```

---

what happens if client crashes?

does server still have a file open?

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

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`

mitigations for blocking

coordinator aborts if still possible

- requires coordinator not to go away
- handles *workers* failing before decision made

workers share outcomes without coordinator

- possibly handles coordinator failing (if all workers still working fine)
- other worker can say “coordinator said ABORT/COMMIT” (even if coordinator now down)
- if any worker agreed to abort, don’t need coordinator