10/20/10

No office hours today
I have to leave at 2pm.

Page Replacement Policies

"Global" Choices

① If a page is dirty/non-dirty
- ~~dirty~~

② If the page is dirty ⇒ I could maintain a cache of memory pages not allocated to user programs ⇒ "safety net"
- instead of putting the dirty page to the swap file, I use a "cached" page.

Specific Policies

1. Random
   - \+ Simple
   - − Not based on usage

2. FIFO
   Select the page that's been in memory the longest
   - − Not based on usage

3. Optimal
   Replace the page that won't be needed for the longest time
   - − Not implementable
   - \+ Used for comparisons with implementable policies

Aging Algorithms

Replace the "oldest" page

- We have a way to measure "age" (length of time since a page was accessed)

Clock- 1 bit

Second Chance- 2 bits

Several algorithms- 4 or 8 bits

Least Recently Used (LRU)

- typically 32-bits

Remote Procedure Calls

Based on the concept of a local procedure call

- "jump" to another section of code
- execute it
- return to calling address & continue - may return a value

The difference with RPCs - the jump is to code on another computer

How to implement RPCs?

① Require user programs to explicitly issue Send/Receive syscalls

+ Easy OS implementation

− Requires the application to "know" the server location
  - & possibly the protocol

Limit flexibility

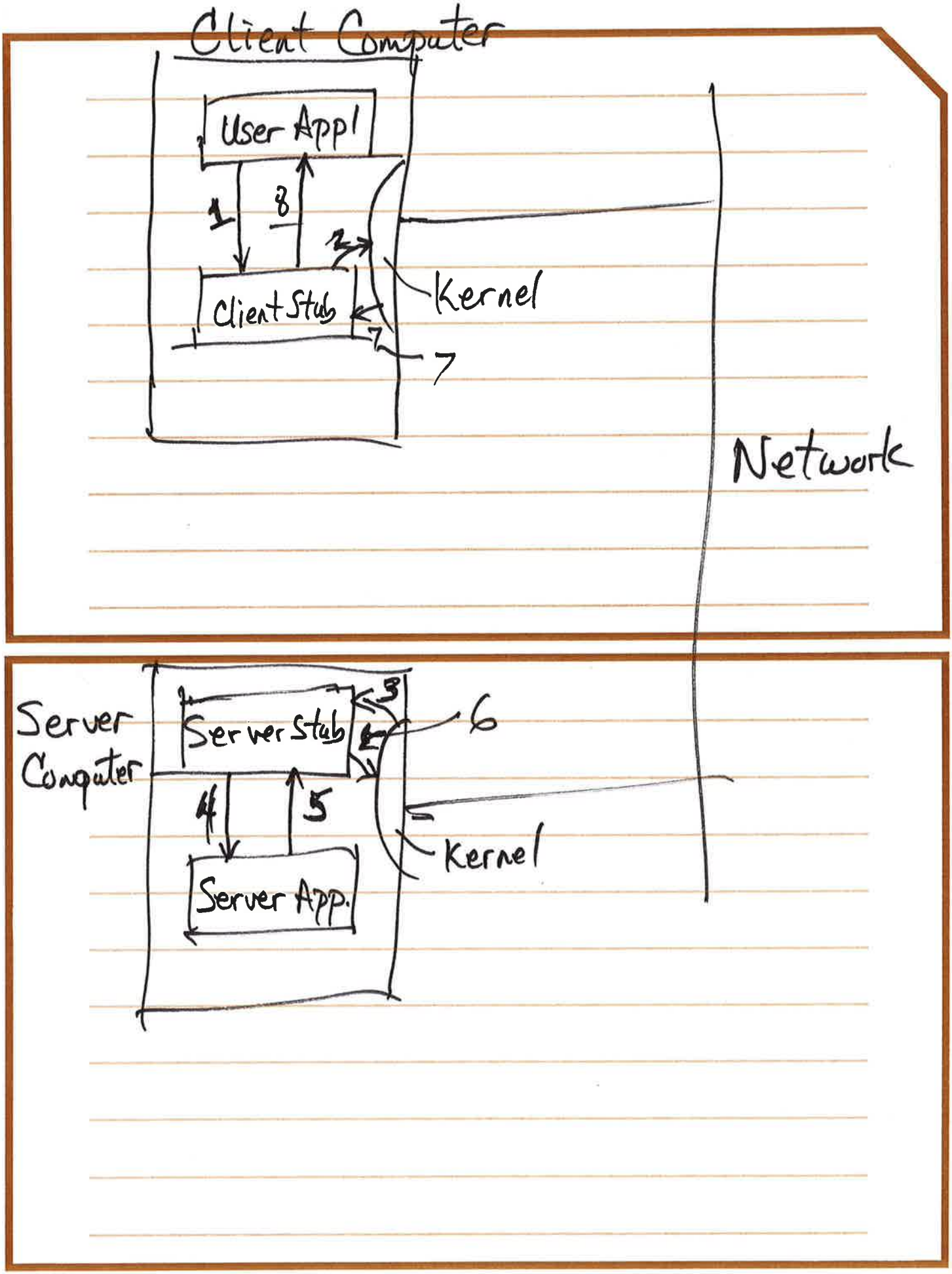
(2) Hide the fact that work is being done remotely from user programs

- No networking code in the user program
- Send/Receive work done by a "middle layer", or by OS kernel.

How to do this?

Use "stub" programs to do the networking

- client stub (issues requests)
- server stub (receives requests)


Client Computer
User Appl
1
8
2
Client Stub
Kernel
7
Network
Server Computer
Server Stub
3
6
4
5
Server App.
Kernel

Where do stubs come from?

Key Idea: Client stub & server stub MUST "understand" each other

Their primary task is to create, parse, send, & receive msg's between each other

Solution: Have a program to generate the stubs

⇓

Stub Generator

Examples: Web Services ⇒ Axis wsdl2java

CORBA - IDL

idl2java

Specification Data

- Request type
- Request version (optional)
- Order of data in msg
- Format & type of each data item

One Last Issue

How does the client stub "know" where the server stub is located?

① Could hard code the network location of the server stub in the client stubs.

Static Binding ⇐ — not flexible

② Dynamic Binding

Client Stub obtains the network location @ request time of Server Stub

Need a new program: Binder

① Track location of server stubs

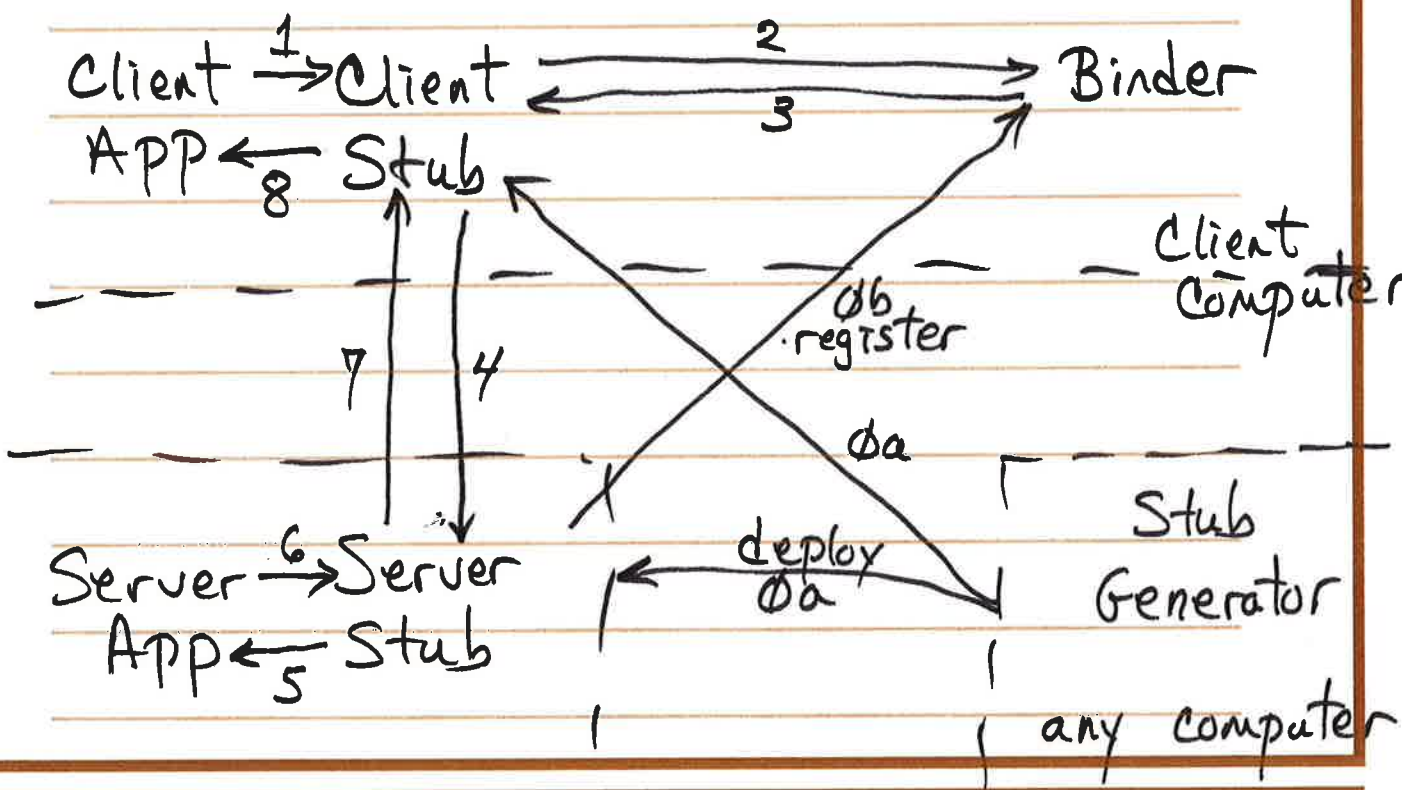
② Respond to client stub requests

Server Stubs "register" with Binders

New Capabilities

- Deregistration msg
- Binders can "poll" ServerStubs
- Load balancing
- Authorization

Dynamic Binding Schematic



Advantages

- Flexible
- Achieved transparency to the user app's

Disadvantage

Complexity

Project 3 Part 3 RPCs

You will have:

Client App ⇒ Nachos User Program
Client Stub → Nachos kernel client

Server Stub } Kernel
Server App } Function

Your server Nachos does not run use programs

nachos -server -m ∅

⇓

runs server kernel function

Client Nachos (Proj 2)

Exception Handler

- Determine the syscall type
- Get the parameters
- Validate " "
- Do the work
- Return a value (or not) to the user program

Proj 3, Part 3 - Client Nachos

- Determine the syscall type
- Get & validate parameters (as much as possible)
- Create & send a request msg to server
- * Wait for the reply msg
- Return a value (or not) - extracted from the reply msg.

Server Nachos

```
while (true) {
    • Wait for a request msg
    • Get request type
        • Must be first in the msg
    • Parse rest of msg
    • Validate parameters
    • Do the work
    • Send a reply msg - maybe
}
```

Scenario- Acquire (on Server)

- If we use real locks, like in Project 2, we won't get proper system behavior

We need a Lock Condition equivalent

- they don't put the server to sleep

ServerLock

- owner
  - machine ID
  - mailbox #
- state - busy/free
- wait queue
  - reply msg

ServerCV

- lock number in Server "lock table" (for the given up lock)
- wait queue
  - reply msg

When waking up a client on a signal, the server "acts" like this client has issued an Acquire

Goal: Share locks/CVs across ALL client Nachos

CreateLock/CreateCV must now have a name

Server must store the name with the lock

Any user program wanting to use a lock must issue a CreateLock

Server will make a new "lock" for the first CreateLock with a specific name.