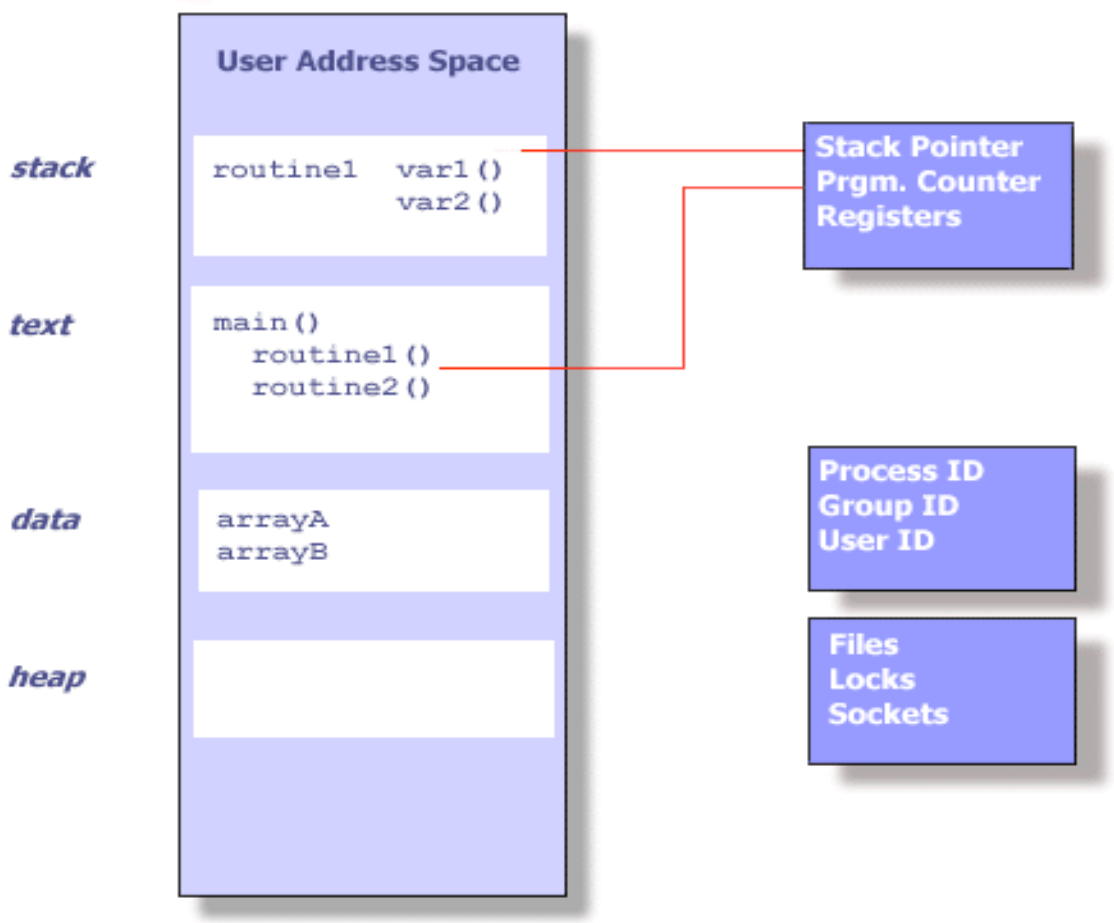
Distributed systems

Programming with threads

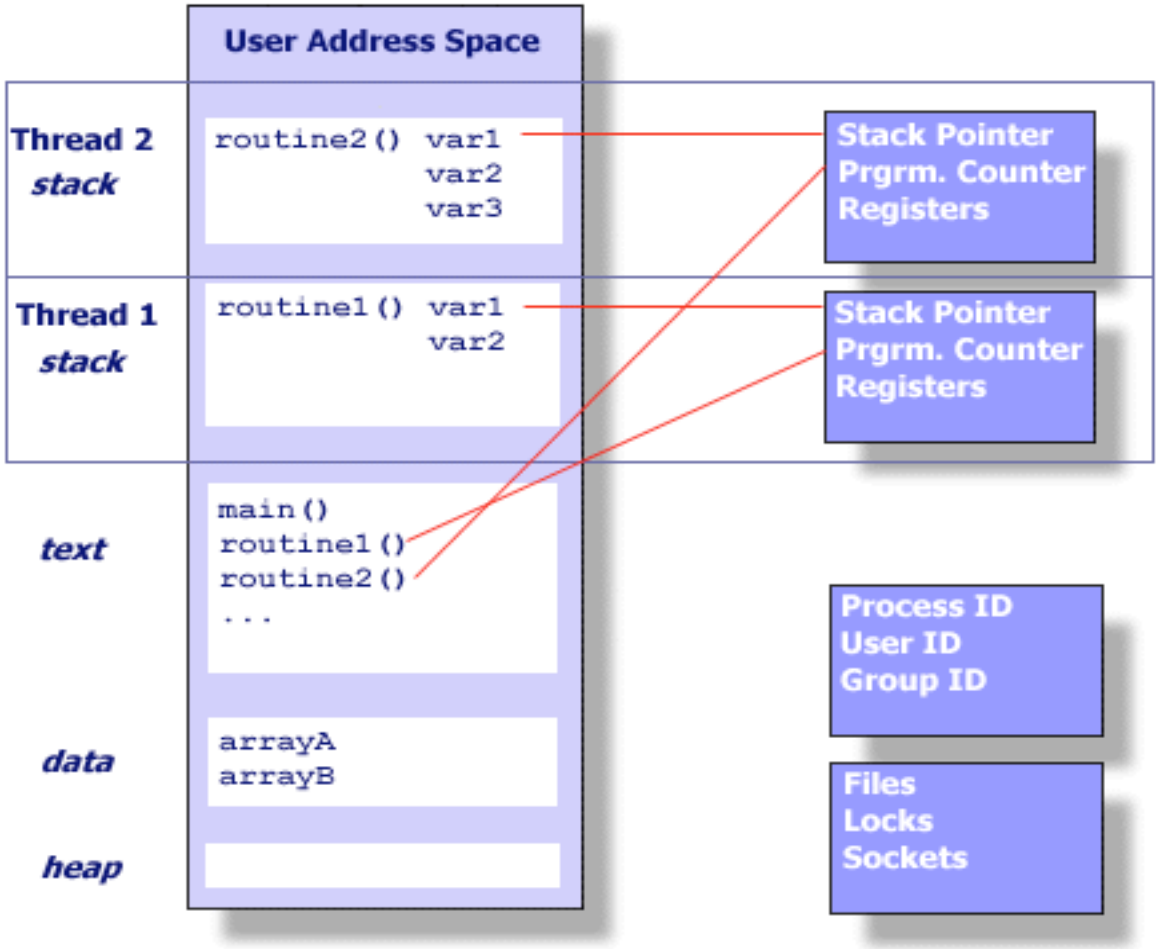
Reviews on OS concepts

- Each process occupies a single address space



Reviews on OS concepts

- A thread of (execution) control has its own PC counter, stack pointer etc within a process.



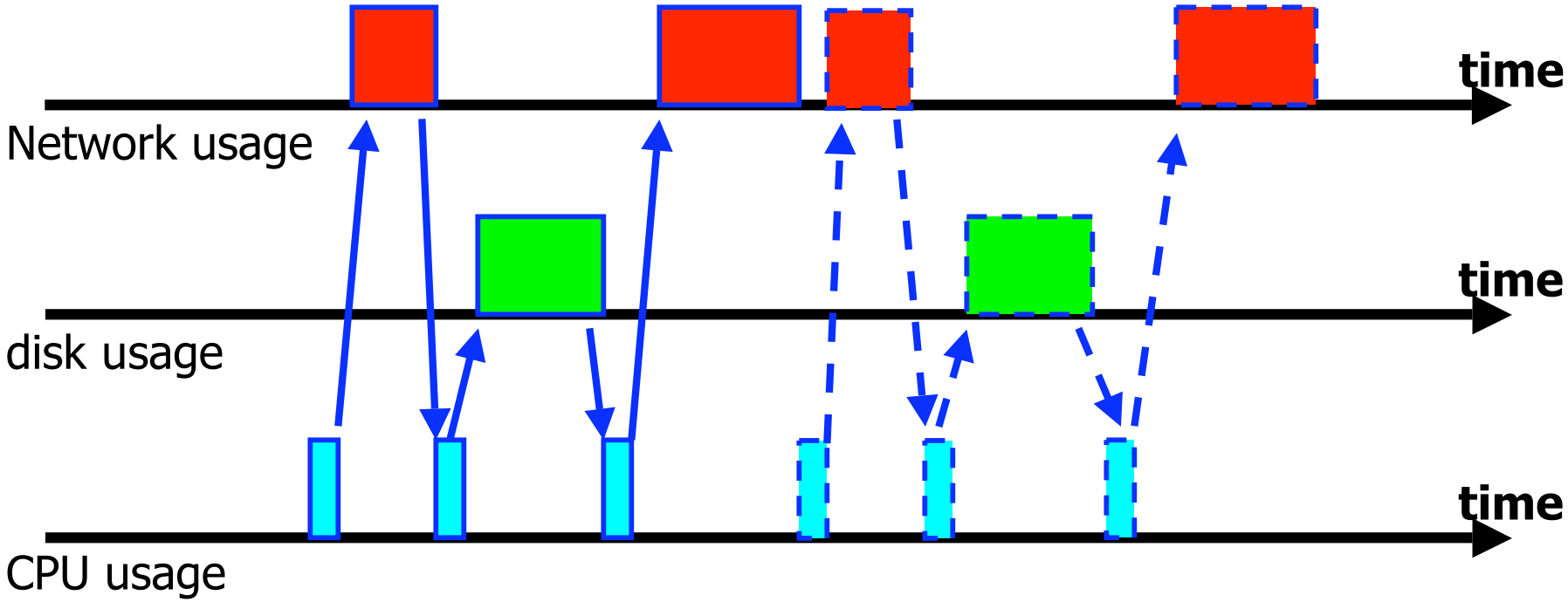
Thread vs. Process

- Why threads?
  - Thread allows running code concurrently within a single process
  - Switching among threads is light-weight
  - Sharing data among threads requires no inter-process communication
- Why processes?
  - Fault isolation: One buggy process cannot crash others

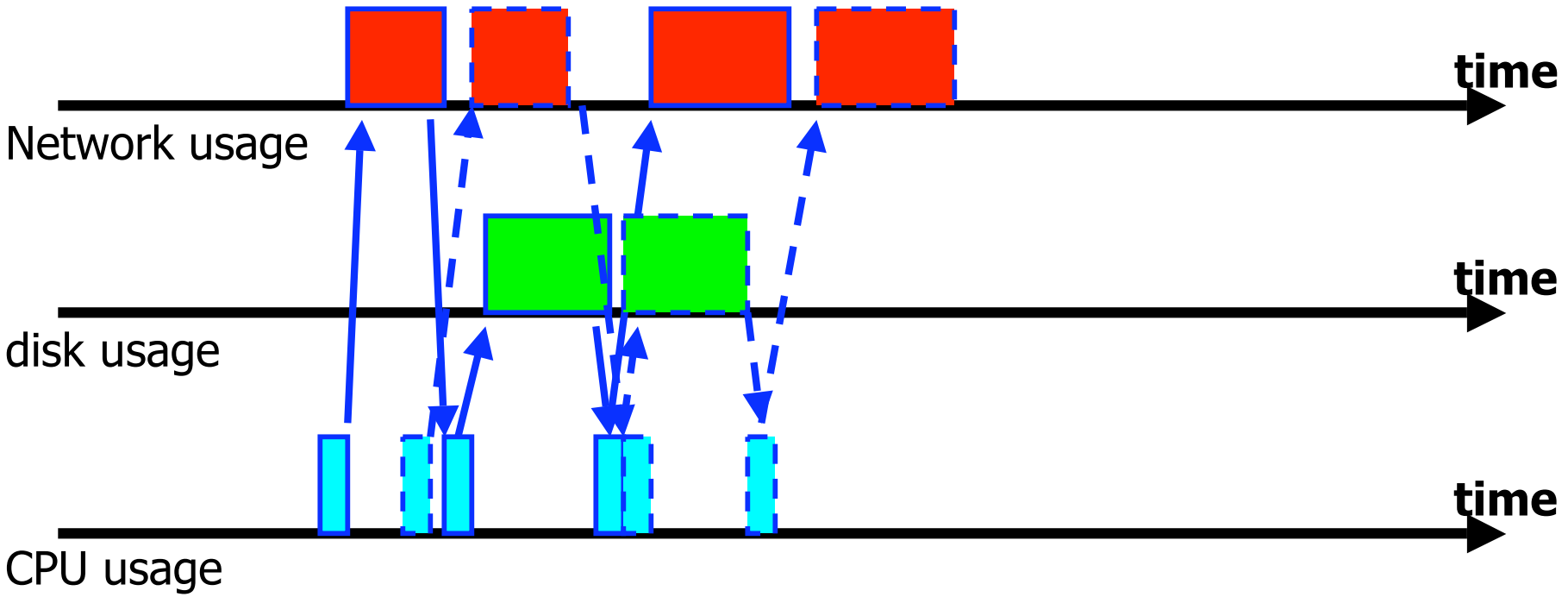
Why concurrent programming?

- Exploit multiple CPUs (multi-core)
- Exploit I/O concurrency
  - Do some processing while waiting for disk (network, terminals, etc.)
- Responsive GUI
  - Respond to users while doing processing in the background
- Reduce latency of networked services
  - Servers serve multiple requests in parallel
  - Clients issue multiple requests in parallel

Single threaded servers do not fully utilize I/O and CPU



Multi-threaded servers achieve I/O concurrency



Designing a thread interface

- Create and manage threads
  - pthread_create, pthread_exit, pthread_join
- Provide mutual exclusion
  - pthread_mutex_lock, pthread_mutex_unlock
- Coordinate among multiple threads
  - pthread_cond_wait, pthread_cond_signal

Common Pitfalls

- Race condition
- Deadlock
  - Better bugs than race
- Wrong lock granularity
  - Leads to race or bad performance!
- Starvation

Remote procedure calls

RPC abstraction

- Everyone loves procedure calls
  - Transfer control and data on local programs
- RPC goal: make client/server communication look like procedure calls
- Easy to write programs with
  - Procedure calls are a well-understood model
  - RPC hides details of passing data between nodes

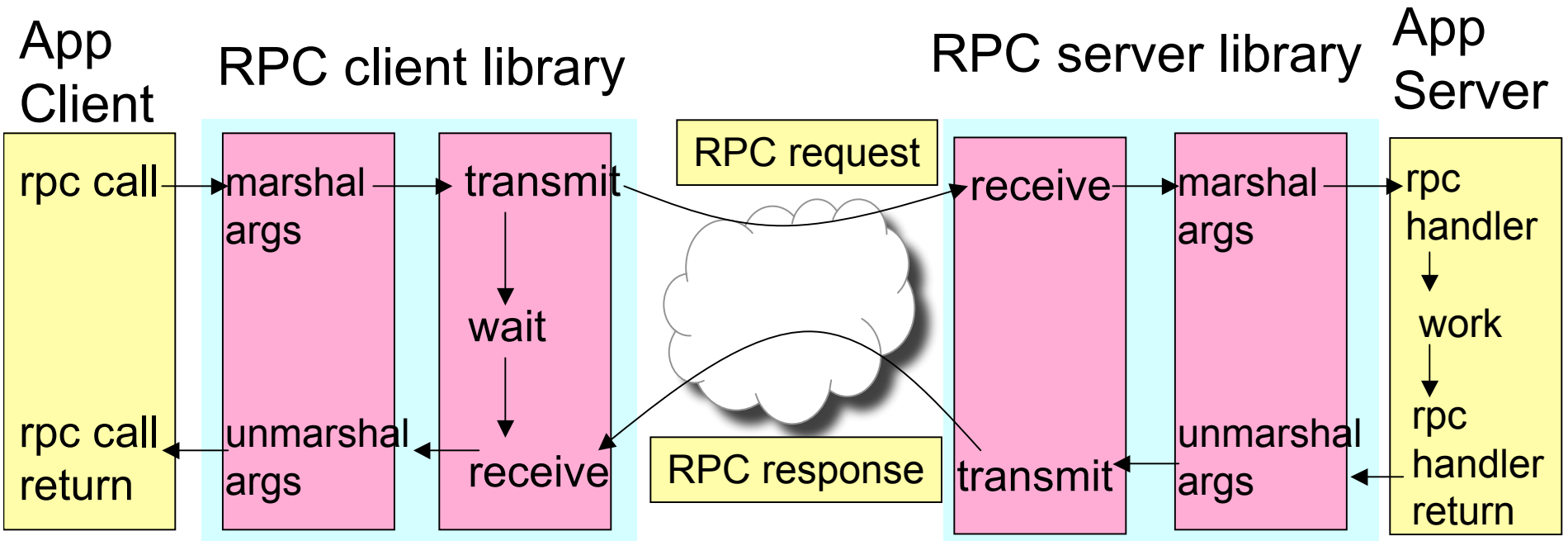
RPC vs. alternatives

- Alternatives:
  - Sockets
  - MPI
  - Distributed shared memory (later classes)
  - Map/Reduce, Dryad (later classes)
- RPC is very popular in programming distributed systems
  - XML RPC
  - Java RMI
  - Sun RPC

RPC architecture overview

- Servers export their local procedure APIs
- On client, RPC library generates RPC requests over network to server
- On server, called procedure executes, result returned in RPC response to client

RPC architecture



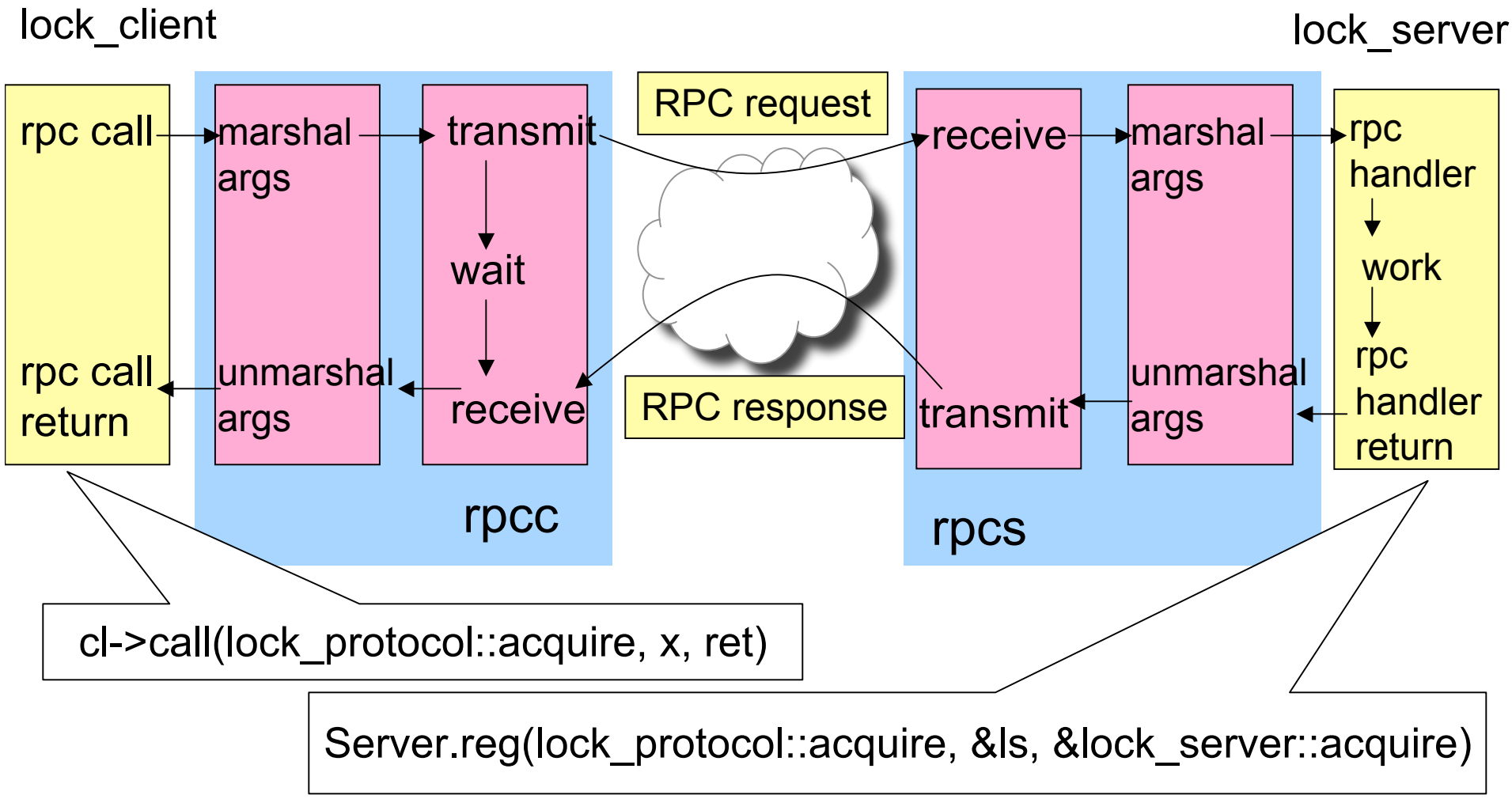
Key challenges of RPC

- RPC semantics in the face of
  - Communication failures
    - delayed and lost messages
    - connection resets
    - expected packets never arrive
  - Machine failures
    - Server or client failures
    - Did server fail before or after processing the request?
  - Might be impossible to tell communication failures from machine failures

RPC failure semantics

- RPC might return “failure” instead of results
- What are the possible outcomes in the face of failures?
  - Procedure did not execute
  - Procedure executed once
  - Procedure executed many times
  - Procedure partially executed
- Desired semantics: at-most-once

YFS's RPC library



RPC semantics

- Does yfs rpc implement at-most-once semantics?
- How would the lack of at-most-once affect applications?

Interactions between threads and RPCs

- Can a client hold locks across RPCs?

```
client_func()
{
   pthread_mutex_lock(&cl_lock);
   cl_rpcc->call(….)
   pthread_mutex_unlock(&cl_lock);
}
```

- Should it do so?

Interactions between threads and RPC

- How about this client side code?

```
client_func()
{
   pthread_lock(&cl_lock);
   for (vector<rpcc>::iterator i = list.begin(); i != list.end(); i++) {
      pthread_mutex_unlock(&cl_lock);
      (*i).call(….)
      pthread_mutex_lock(&cl_lock);
   }
}
```

Interactions between threads and RPCs

- Can a server make a RPC call during RPC handler?