Distributed Systems (Ch 15 and 16 of S&G)

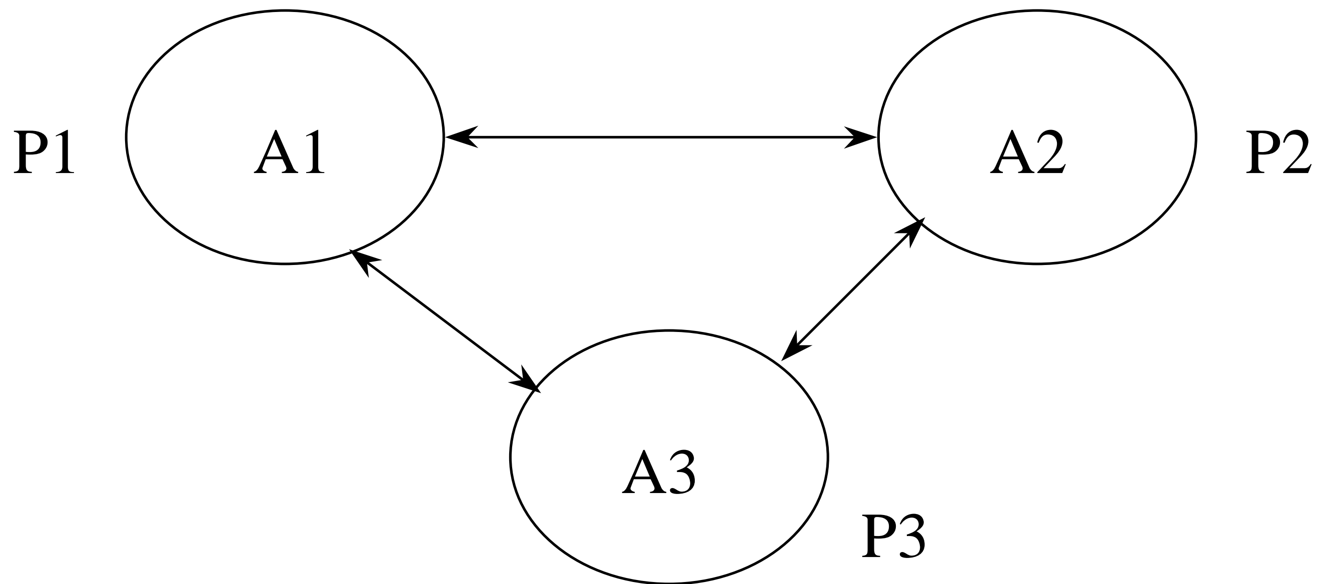
What is a distributed system?

- nodes interconnected by LAN/WAN
  - LAN
    - twisted pair, coax, optical fiber, switches
  - WAN
    - telephone lines, cable, satellites, microwave links, wireless, ...
  - media access protocols
    - ATM, Ethernet, token-passing, point-to-point dedicated links, ...

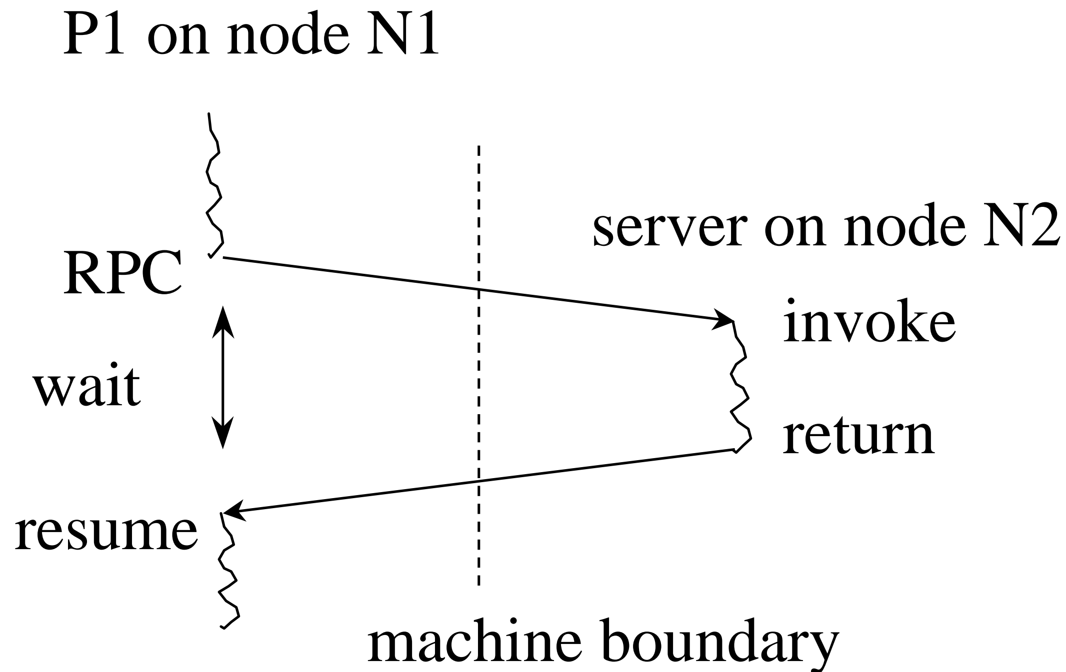
- levels of coupling
  - network operating system (loose coupling)
    - each node autonomous
    - services?
    - issues
      - byte ordering
      - interfacing to local OS at each end
      - transport protocols
  - distributed OS (tighter coupling)
    - nodes cooperate to achieve a single task
    - seamless interface to physically distributed resources
    - issues
      - distribution/migration of data and computation
      - data shipping/function shipping, load balancing

Structure of a distributed system

- A1, A2, A3 are distinct address spaces



- how do they communicate?
  - use messages: send(P2, msg)
  - peer-to-peer messages implemented via message kernel
  - issues:
    - establishing communication port between nodes
    - buffering messages at sender and receiver
    - copying to/from address spaces
    - authentication of messages
  - send/receive interface too unstructured
    - why?
  - how to make it more structured?



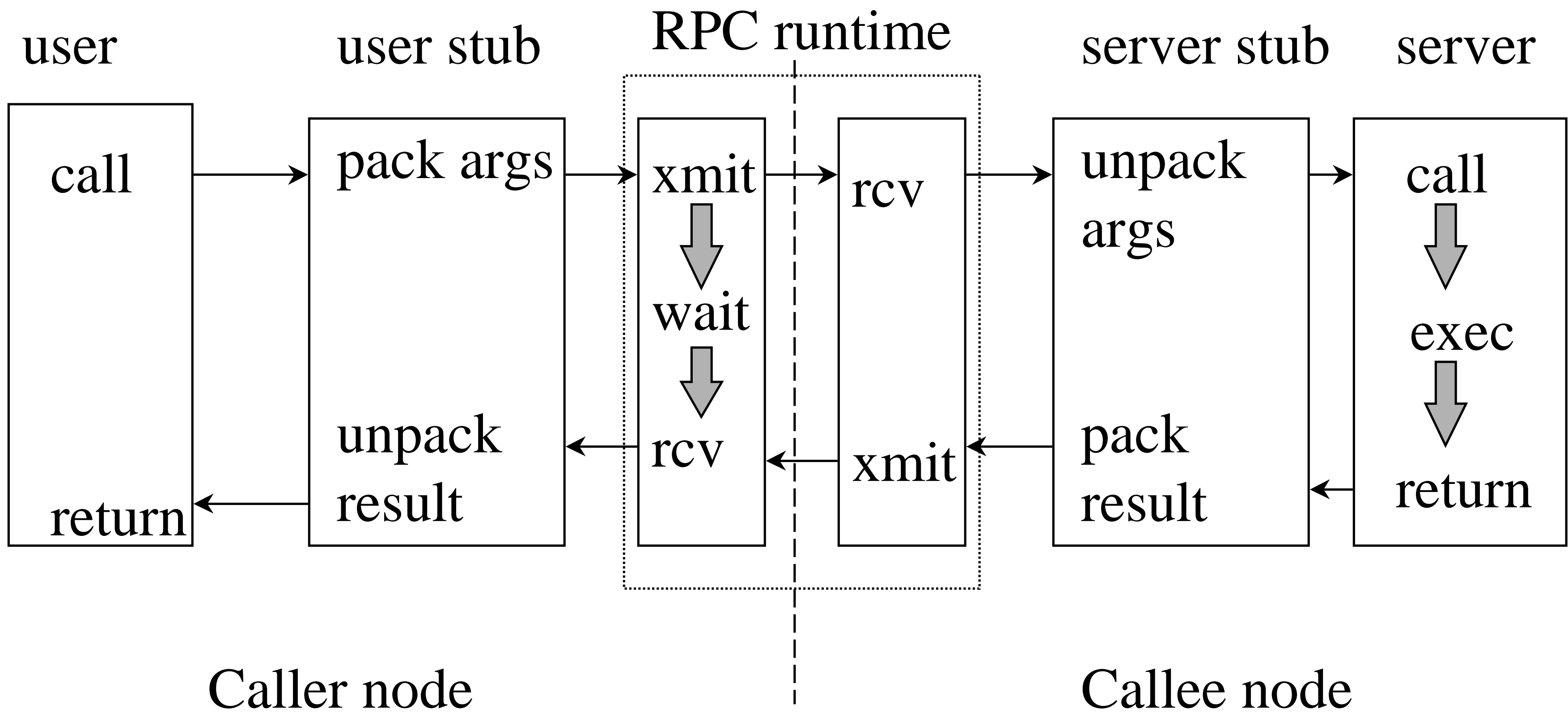
- N1 can run other processes while P1 is waiting on RPC completion
- RPC abstraction is natural extension of sequential programming practices
- what are the issues in implementation?

Issues

- semantics in the presence of communication failures
- parameter passing (pointers,…)
- binding
- data transfer protocols
- data integrity and security
- speed….

Addressing these issues

- failures
  - at most once semantics
- parameters
  - in-out through "stubs" that marshall and unmarshall at client/server
- binding
  - static (fixed at compile time)
  - dynamic (through a name server)
    - registration and lookup (Figure 16.1)



Five pieces of program involved

- user
  - no different from any old program making use of procedure calls
- user stub
  - marshall args; identify target procedure; hand over to RPC runtime to deliver to callee
- server stub
  - unmarshall the args; make a normal procedure call in server using the args
- user and server code part of distributed app
- stubs generated automatically

- writing a distributed computation
  - interface module (API)
  - client code that imports the API
  - server code that exports the API
  - stub generator generates user-stub and server-stub using the API
  - bind user with user-stub; server with server-stub
  - ready for business….
- binding
  - naming (what to bind to, compile time decision)
  - location (where to bind to, a runtime decision)

Naming

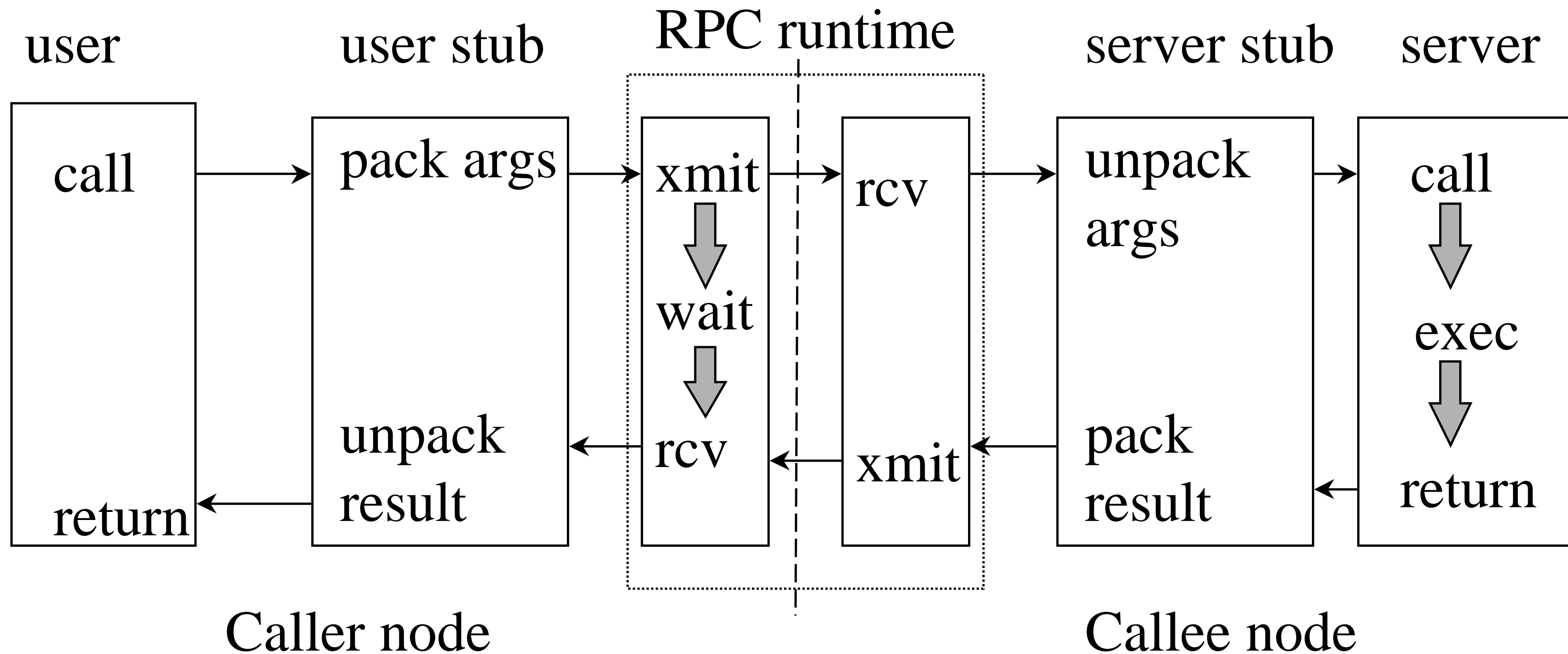
- type
  - generic service (mail xfer, file xfer)
- instance
  - particular server from a set of available ones
- semantics of the interface name outside the scope of RPC package
- location of an instance dictated by the RPC package

Location

- early binding
  - machine address of server hardcoded in the app
- broadcast to locate a server
  - too much interference
- nameserver
  - e.g.
    - type: file server
    - instance: trantor, cleon
  - groups and individual

fileserver	trantor, cleon
trantor	130….
cleon	130….

RPC at Work



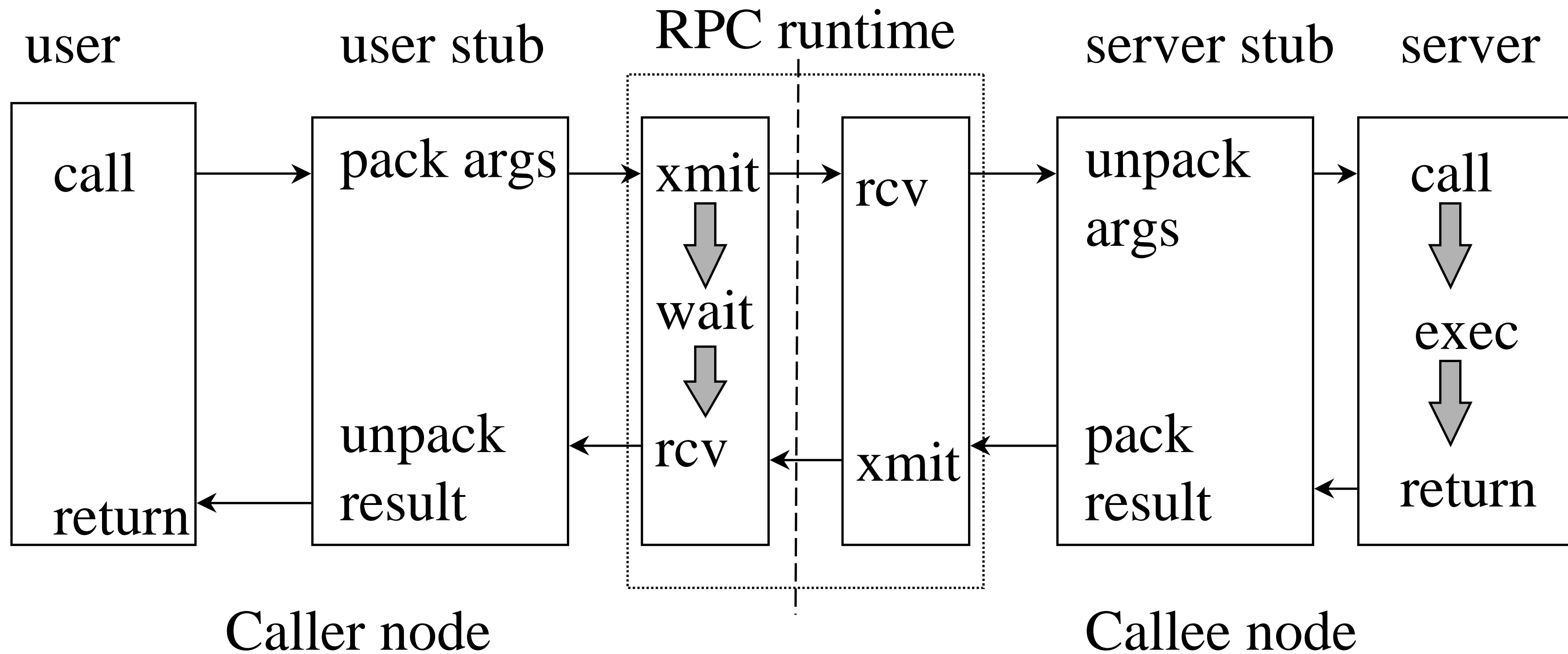
- Prior to first call
  - “export” by server
  - “import” by client => binding takes place

RPC Server

- Upon creation
  - register “name” of RPC with the RPC runtime using the “export” mechanism
  - RPC runtime informs name-server of this service
  - RPC runtime at server now ready to service RPC calls with this “name”
- Multithreaded
  - usual to maintain “pool” of threads to avoid thread creation overheads
  - no thread creation overhead unless server gets extremely busy and has to spawn new threads

RPC Clients

- Upon startup
  - register intent to use RPC call via “import”
  - RPC runtime at client runs around does the binding
  - client makes a normal “call” after “import”
- To make an RPC call
  - client makes “RPC” call
  - user-stub creates a call packet
    - marshalled parameters, “name” of RPC
  - RPC runtime at client translates “name” to server address and sends call packet to server



- Steady state
  - RPC serves potentially multiple clients
  - does the server need to maintain any "state" about the clients it is interacting with?
  - if yes, what "state"?

At Most Once Semantics

- client crash and restart
  - RPC at the client gives a new "incarnation id" to the client
  - client has to "rebind" to the service
  - server uses the "client id" to distinguish this instance of the client from the previous one
- server crash and restart
  - server gets a new "server id"
  - all clients bound to the previous incarnation are out of luck…they have to "rebind"

Using RPC

- build higher level distributed services
  - NFS
  - SMTP
  - Telnet
- web servers
- database engines
- peer-to-peer servers