Evolution of Distributed Computing

- Evolving fast since early 1970s
  - Milestone event: ARPANET (BBN, 1969), first internet
  - Not always clear what should and would be distributed (which resources): apps, files, memory, processing
  - Key enablers: **local** and **wide-area** networking
- First stage
  - Sharing files: ftp
  - Remote access to CPUs: telnet
- Ftp is a pain, ergo distributed file systems
  - Same system call (fopen) works local and remote
- Distributed concurrent processing: divide and conquer across machines
- Remote execution: call non-local procedure/method
  - Gave birth to client-server paradigm
  - Remote procedure call: RPC …. then remote objects

Distributed System Architectures (1.3)

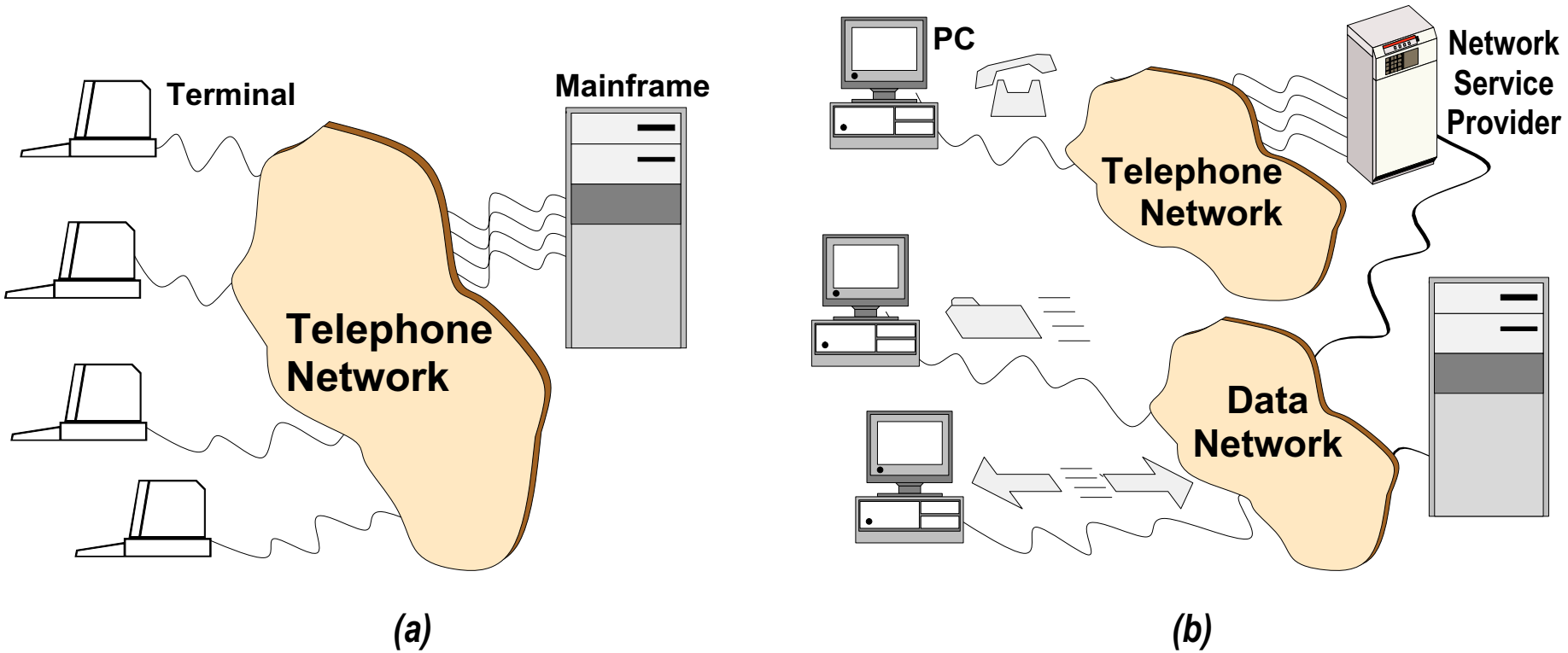
- Q: what is an architecture?
- Architecture vs. framework
- DS architectures have evolved as hardware and software available became more sophisticated
- Presented now in rough order of evolution
- High-level architectures given, many ways to implement (we will study some….)

Remote Access

- The most basic form of distribution
- Goal: provide distributed access to centralized facilities

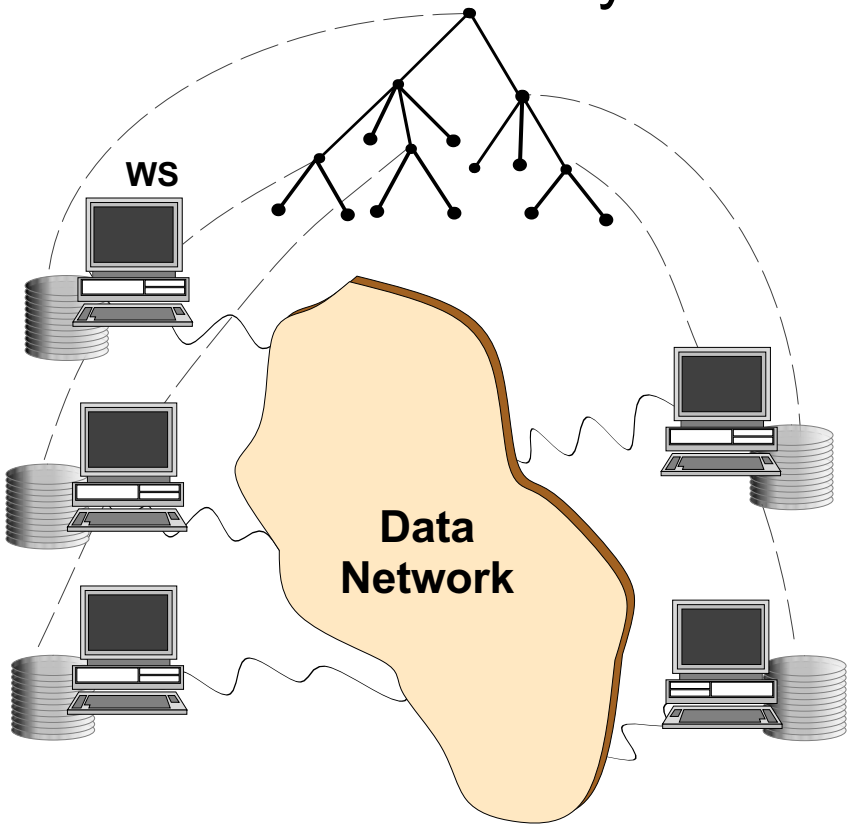
(a) terminal access over POTS: most primordal

(b) remote sessions over data network: files, email, …

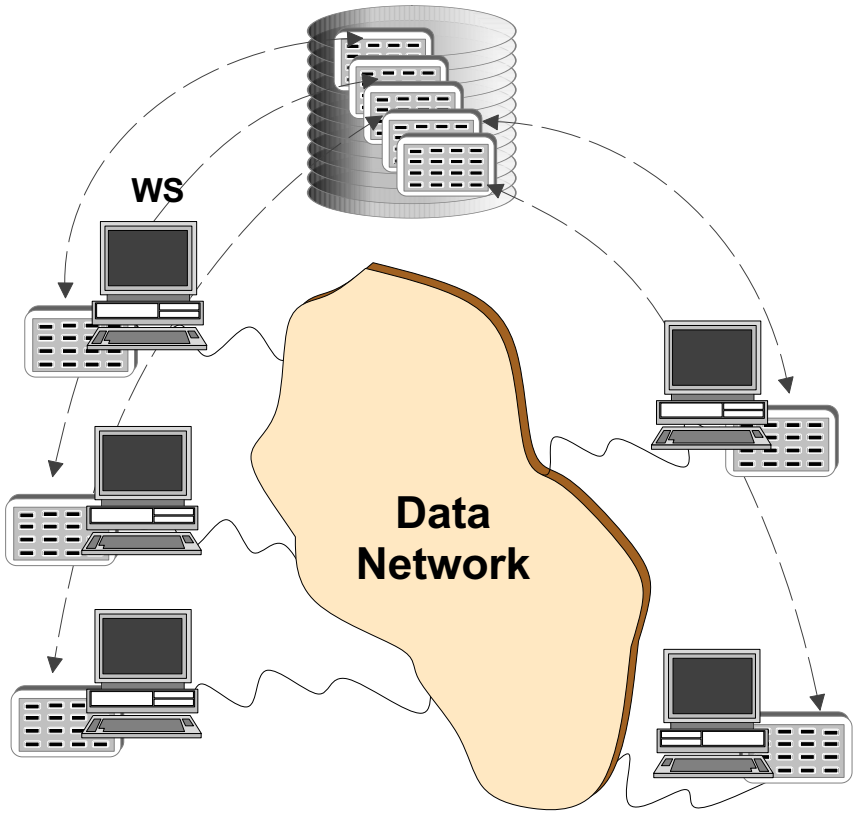


File and Memory Distribution

- Coming of workstations allowed computing power to be spread out
  - Added up to a lot of resources
  - File and memory distribution share on a peer basis



(a)



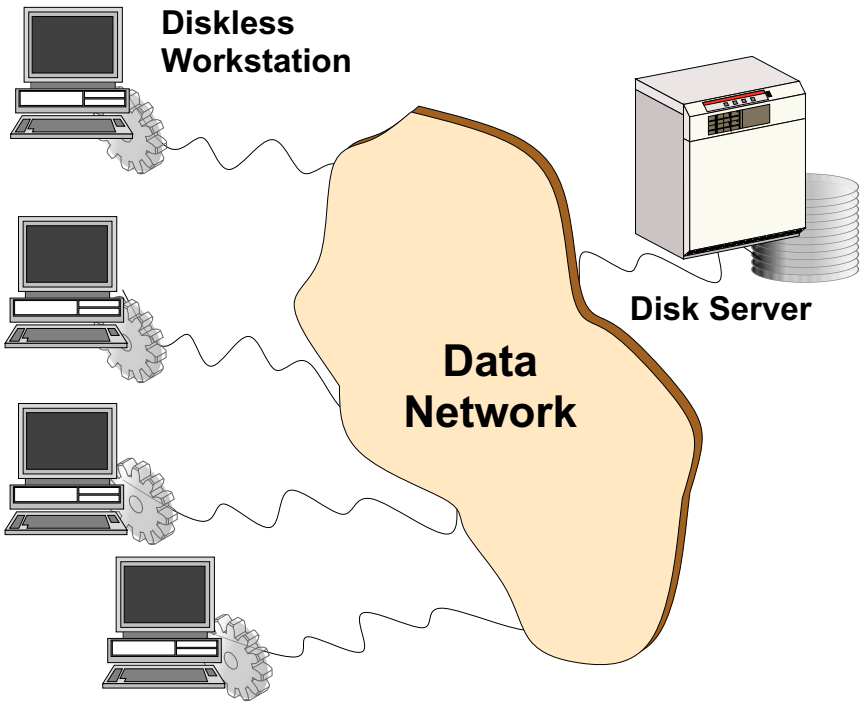
(b)

Remote Access II

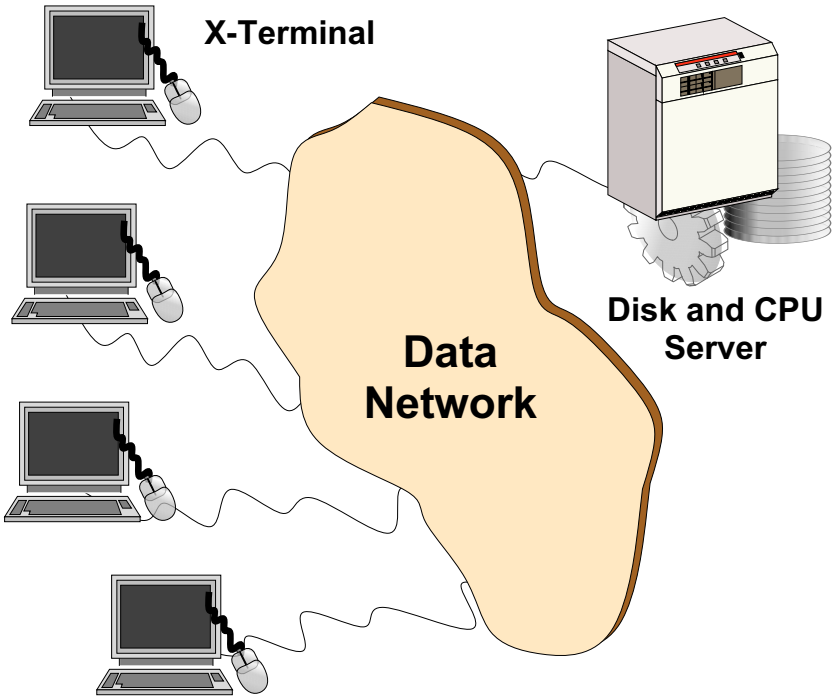
- Oops: DSs not so easy to administer & share perfectly….
- Backlash: re-centralize some resources

(a) Diskless workstations

(b) X-Terminals



(a)



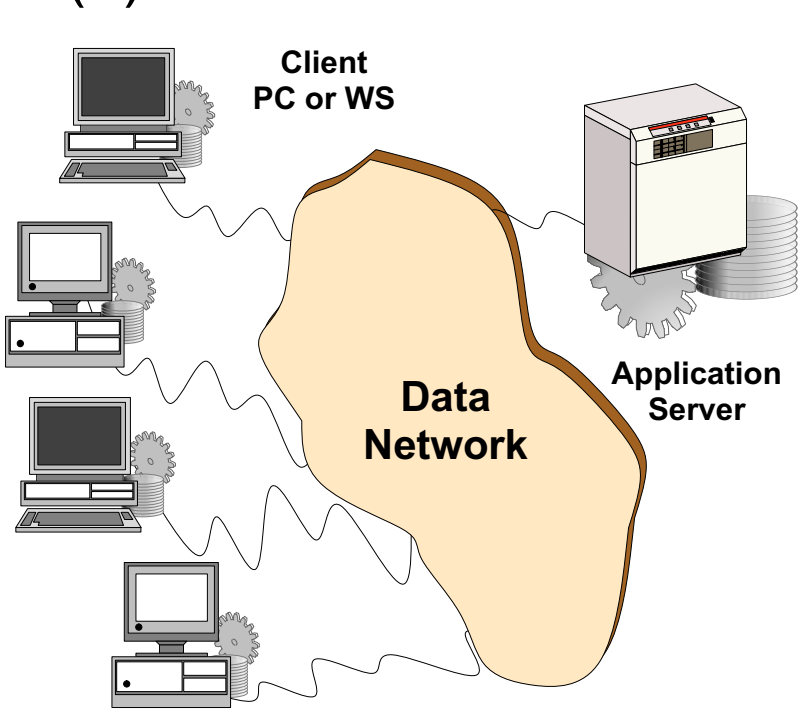
(b)

Client-Server Architectures

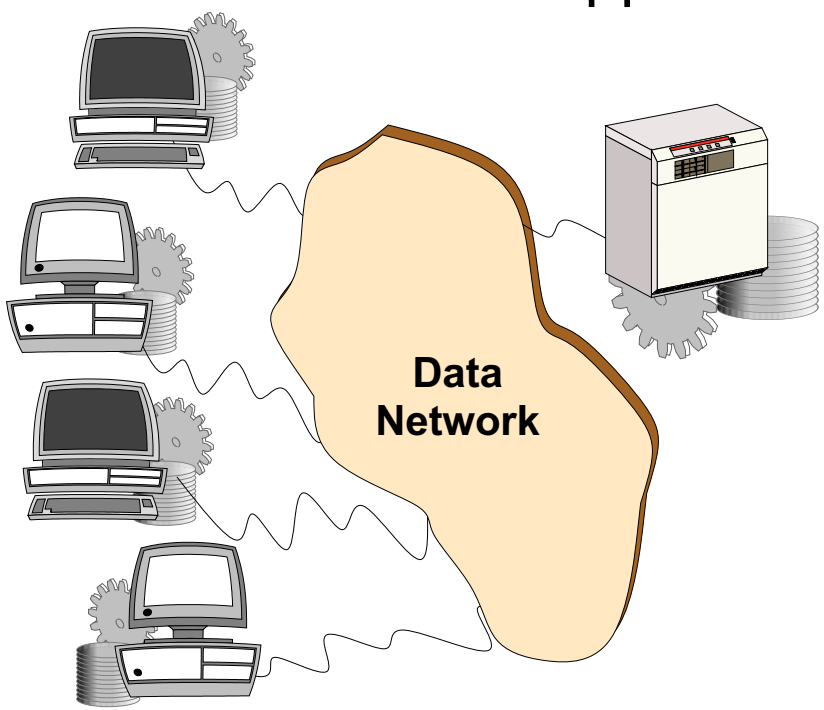
- Oops: X-terminals and diskless WSs unsatisfactory
- Client-server architectures: break up application into two remote pieces
  - Most widely-deployed architecture today

(a) Client-Server Architecture

(b) Fat clients: client machines have code for each app.



(a)

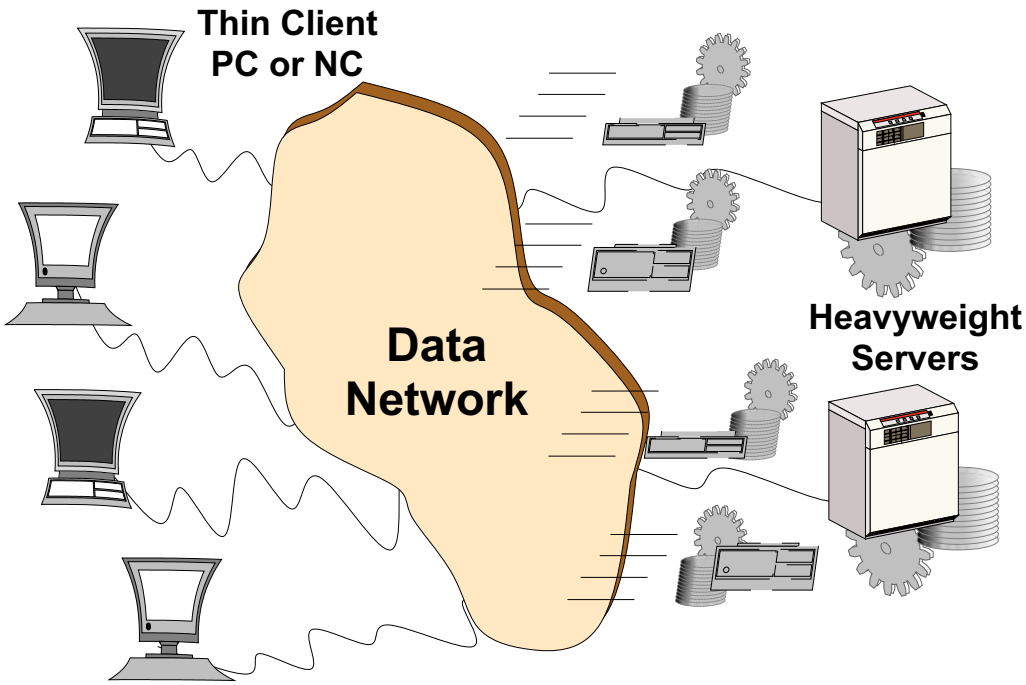


(b)

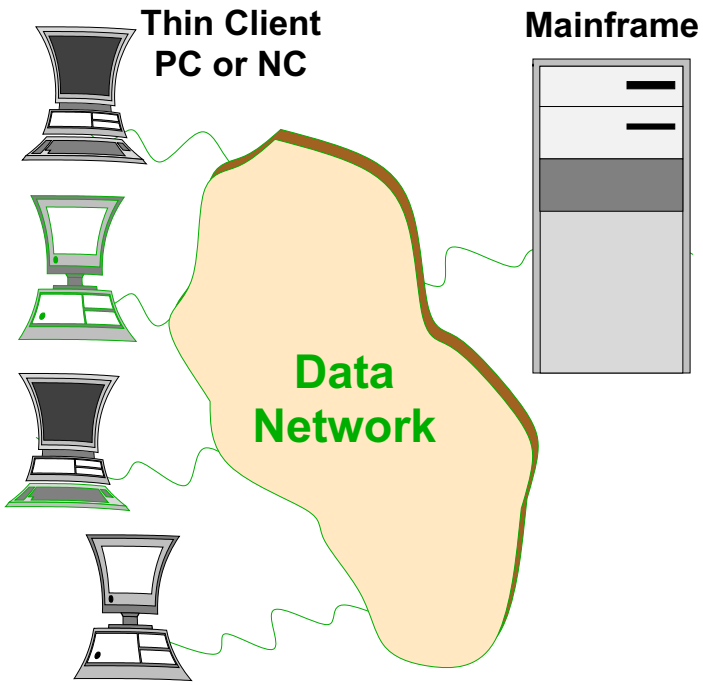
3-Tier Client-Server Architectures

- Oops, fat clients a big pain (code, files, config)
- Oops, too much distribution at server level
- Thin clients: reduced functionality (but more than X-Terminals)
- 3-Tier architecture
  1. Human-client interface (GUI machine)
  2. Application server
  3. Database or secondary servers “behind” app server

3-Tier Client-Server Architectures (cont.)



(a)



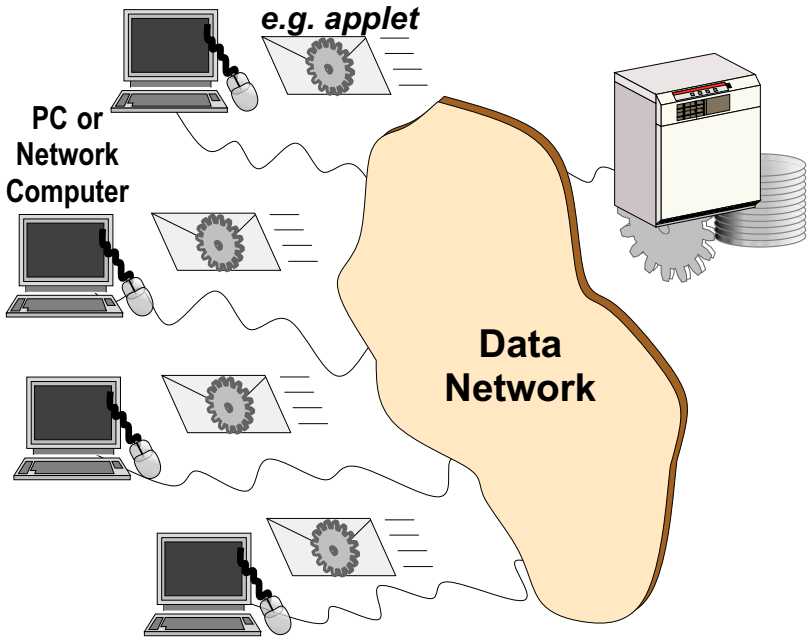
(b)

Mobile Code and Site Architectures

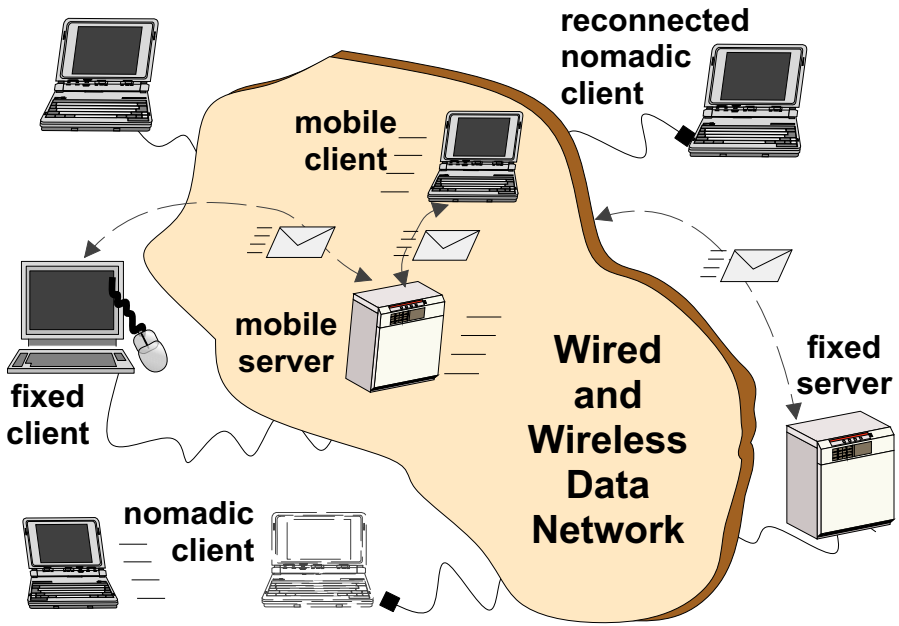
- Oops:
  - thin clients too limited
  - non-dedicated cycles
- Mobile code: ship code when needed to client

  (a) Portable and mobile code

  (b) Mobile Site Architectures: clients and servers can move



(a)



(b)

Mobile Site Architectures (cont.)

- Lots of possible applications
  - military
  - emergency networks
  -
- Variations
  - Only allow client to move (mobile cell phone networks)
  - Only allow sites to move while offline, then reappear ("nomadic computing")… very popular today, growing wildly fast…

Event-Based Architectures

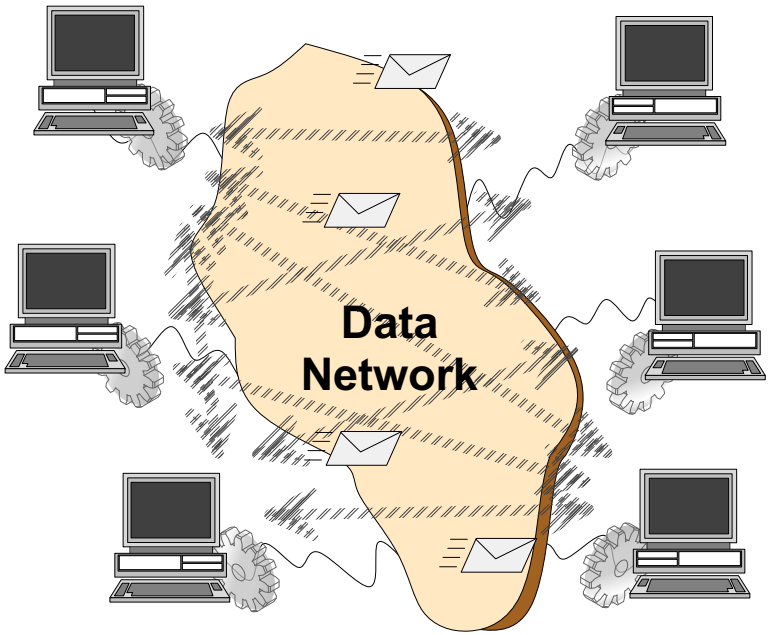
- Oops: client-server paradigm does not represent all kinds of useful distributed applications
- Some are more peer-peer
- Some require handling unsolicited events
- Example: client server
  - client: initiates request, gets reply
  - server: maintains state, gets request, sends reply based on state
- Event-based (AKA message-based) architectures
  - Server generally maintains state, but can initiate events / messages / invocations
  - E.g., when its state crosses a threshold
  - Examples?

Event-Based Architectures (cont.)

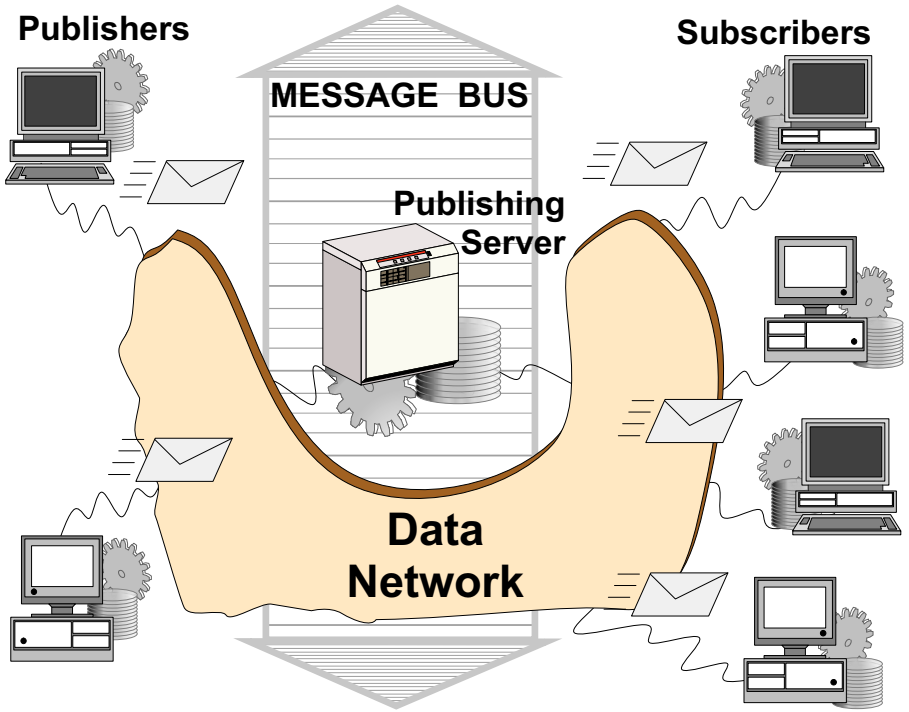
- Event-based communication often multicast/group
- Can run multipeer conversations: directly sending messages to another
- Asymmetric example: producer-consumer
- Example apps: email, USENET news
- “Push” technologies are producer-consumer
- Enhancement: let producers and consumers not have be online at same time: message queues
- Enhancement: let consumers specify (subscribe) only what (type,value) of messages they receive: “publish-subscribe” (AKA “message bus”)

Event-Based Architectures (cont.)

- Figures
  - (a) Multipeer
  - (b) Publish-subscribe



(a)



(b)

Formal Notions for DSs (1.4): DS Models

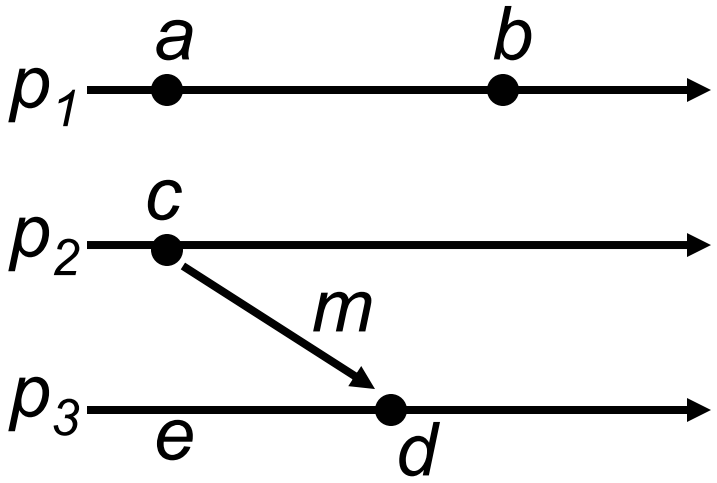
- Warning: heavy notation ahead
  - Only a small part of the class!
- DSs modeled as set of *N* processes or participants *p* residing on *M* processors or sites denoted *s*.
- Sites interconnected by some network, topologies vary
- Evolution of a system modeled by events e^i_p
  - i^{th} event of timeline of each process *p*
  - Omit subscript or superscripts when not ambiguous
- When needed: associate physical timestamps with events: *t(e)* is the real time instant when *e* took place
  - Alternately, t_0, t_a, t_b, …

DS Models (cont.)

- State of a process, *S*, is modified upon the occcurence of each event
- Evolution of state of *S* is its history, *H*
- *H* is an ordered set of tuples: <event, event state>
- A run is an ordered set of events in a process execution, described by a history
- A distributed run is a partially ordered set of events in the execution of several processes

Representing Distributed Computations

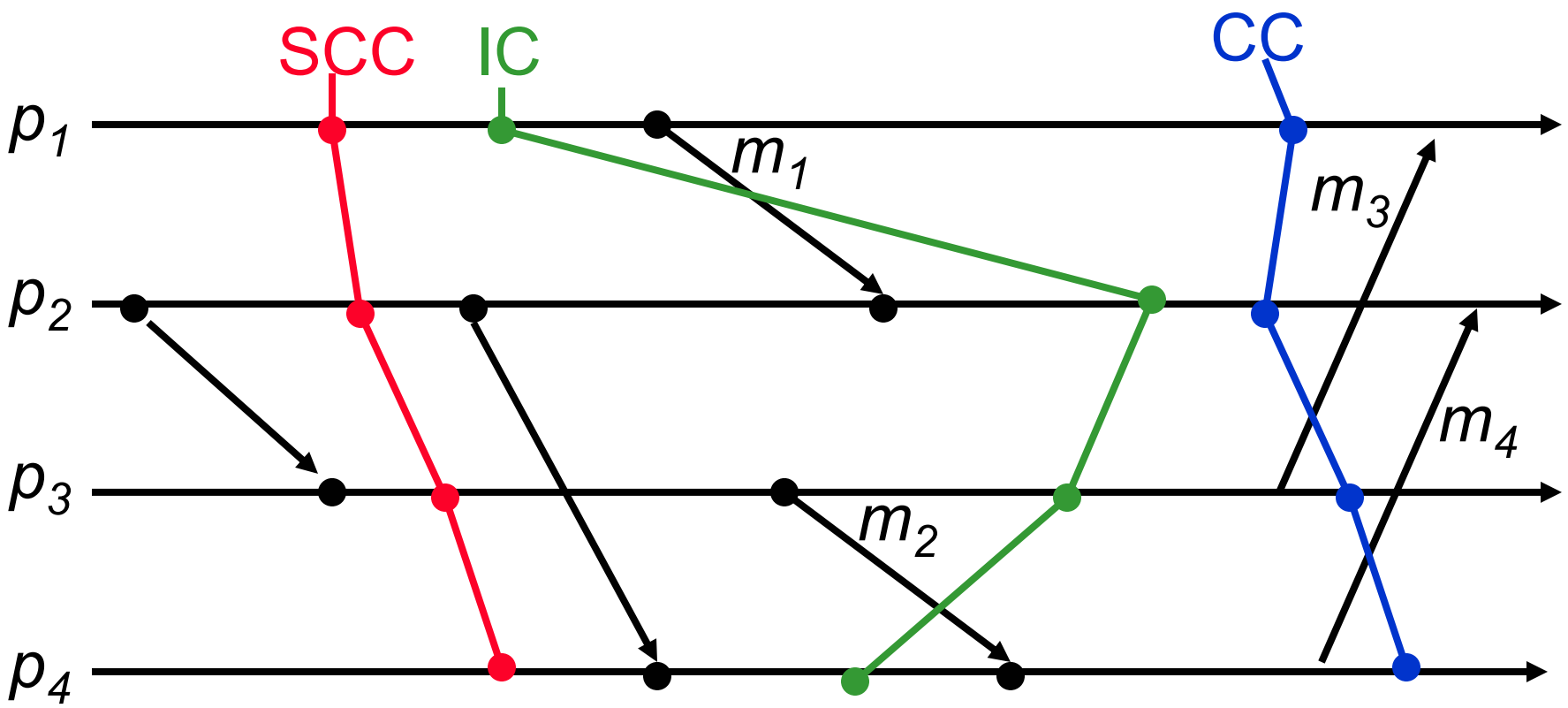
- Events at a process can be
  - execution events: internal computations
  - send events: sending a message to another process
  - receive events: receiving a message from another process
- Message exchanges depicted with timelines: e.g.



Global States

- Sometimes very useful to get a global "picture" of a distributed system
- Global state (GS) of a DS at any point is a vector of its individual process states: $S = \{S_1, S_2, \ldots, S_M\}$
- Two viewpoints of system evolution:
  - Interleaving view: system goes through a succession of states
  - Space-time view: system goes through a partially ordered set of events occuring in several processes in the system
- A cut (in space-time view) is a segment intersecting the timelines of all processes.

Global States (cont.)



- Strongly consistent cut (SCC): faithfully represents GS of the system
- Inconsistent cut (IC): gives invalid picture of GS
- Consistent cut (CC): gives valid but possibly incomplete picture of the GS of the system

Distributed Systems Foundations: © 2001-2003 Carl Hauser & David E. Bakken. Some figures © 2001 Verissimo and Rodrigues

DS Properties

- Goal: specify a system with high-level properties
- Safety properties: something bad (wrong events) never takes place
  - Specification: predicate *P* will never be true in the DS
- Liveness properties: something good (positive event) eventually takes place
  - Specification: predicate *P* will eventually be true in the DS
- "any delivered message is delivered to all correct participants": safety property
- "any message sent is delivered to at least one participant": liveness property
- Timeliness properties relate the times at which events occur – e.g. event a occurs < .1 second after event b.