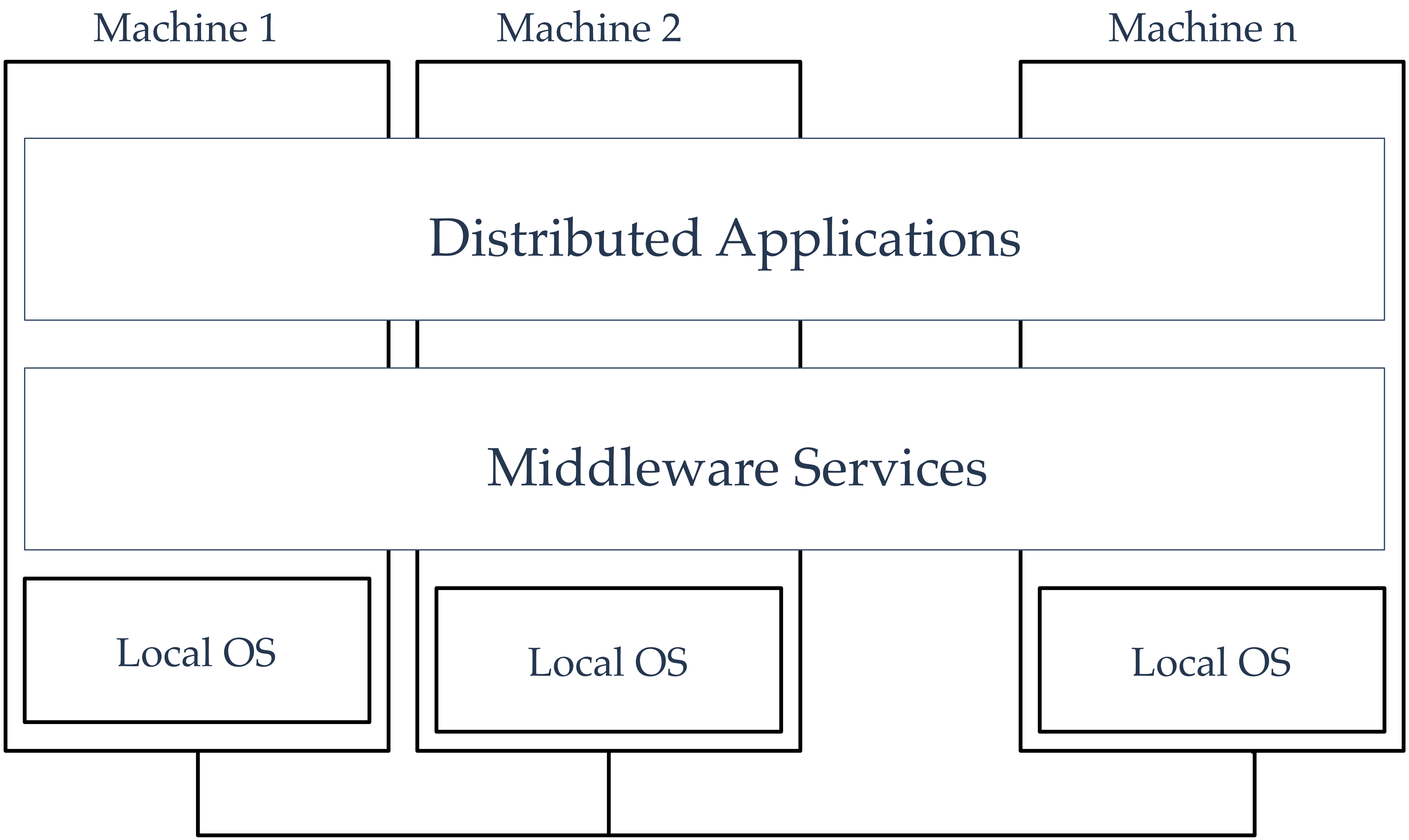
Module 2
Distributed System Architectures

System Architecture

- Defines the structure of the system
  - ➡ components identified
  - ➡ functions of each component defined
  - ➡ interrelationships and interactions between components defined

Software Layers



Layers

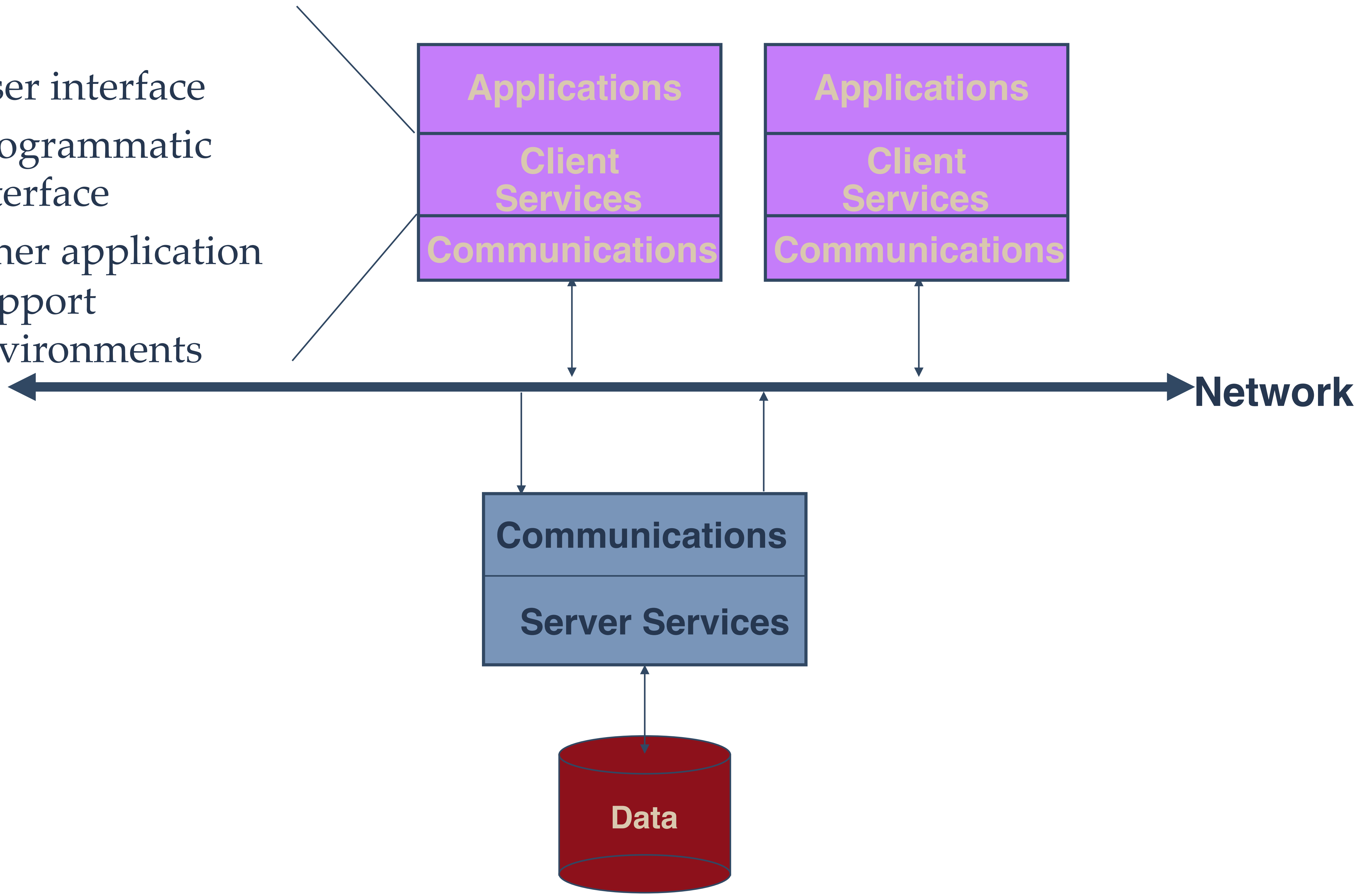
- Platform
  - ➡ Fundamental communication and resource management services
  - ➡ We won't be worried about these
- Middleware
  - ➡ Provides a service layer that hides the details and heterogeneity of the underlying platform
  - ➡ Provides an "easier" API for the applications and services
  - ➡ Can be as simple as RPC or as complex as OMA
    - ✦ RPC (Remote Procedure Call): simple procedure call across remote machine boundaries
    - ✦ OMA (Object Management Architecture) : an object-oriented platform for building distributed applications
- Applications
  - ➡ Distributed applications, services
  - ➡ Examples: e-mail, ftp, etc

System Architectures

- Client-server
  - ➡ Multiple-client/single-server
  - ➡ Multiple-client/multiple-servers
  - ➡ Mobile clients
- Multitier systems
- Peer-to-peer systems

Multiple-Client/Single Server

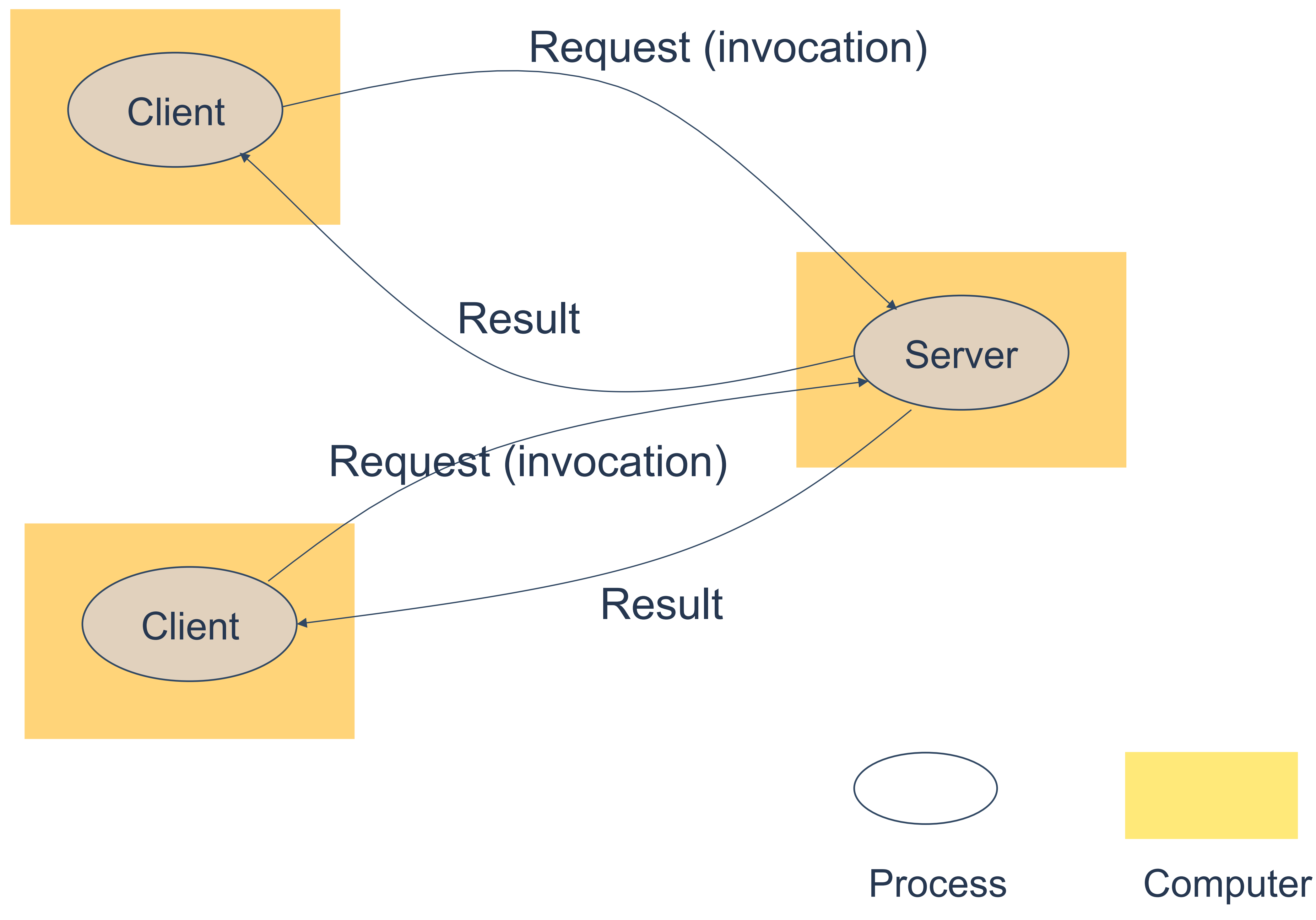
- User interface
- Programmatic interface
- other application support environments



Advantages of Client/Server Computing

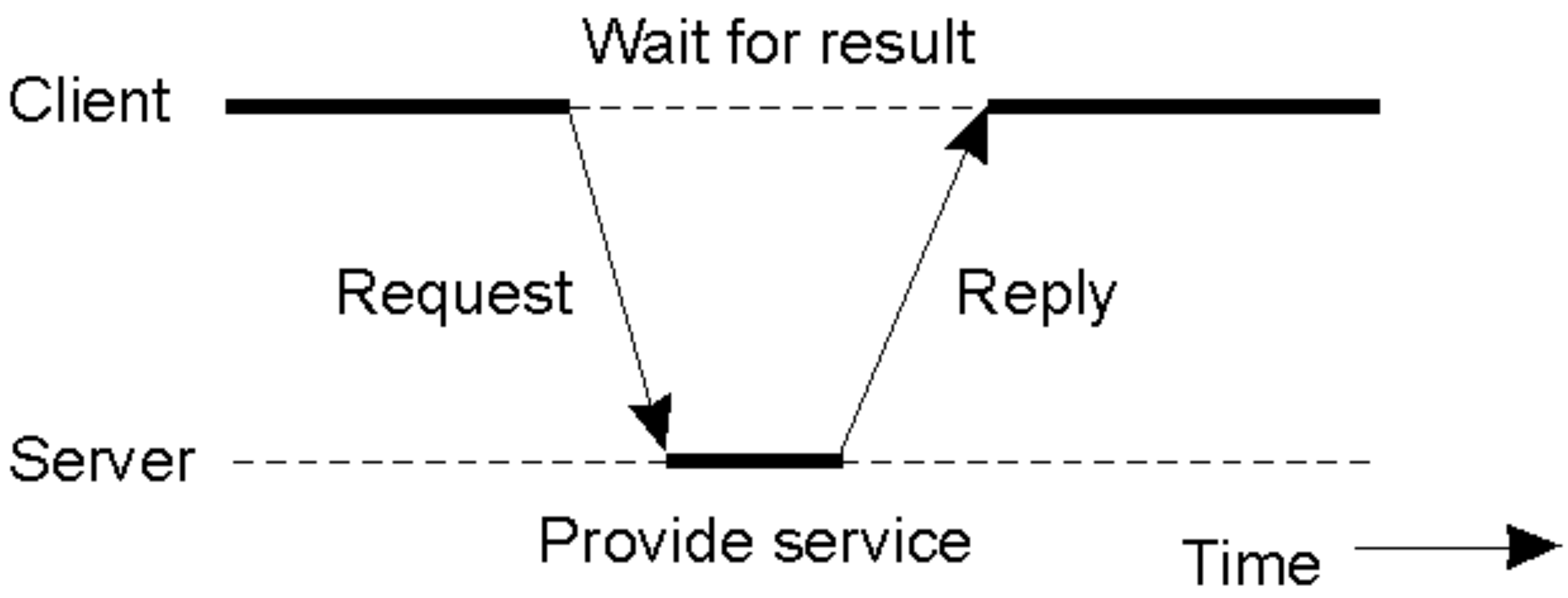
- More efficient division of labor
- Horizontal and vertical scaling of resources
- Better price/performance on client machines
- Ability to use familiar tools on client machines
- Client access to remote data (via standards)
- Full DBMS functionality provided to client workstations
- Overall better system price/performance

Client-Server Communication



Client-Server Timing

- General interaction between a client and a server.



From Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e
© Pearson Education Inc., 2007

An Example Client and Server (1)

- The *header.h* file used by the client and server.

```
/* Definitions needed by clients and servers.              */
#define TRUE            1
#define MAX_PATH        255     /* maximum length of file name          */
#define BUF_SIZE        1024    /* how much data to transfer at once    */
#define FILE_SERVER     243     /* file server's network address        */

/* Definitions of the allowed operations */
#define CREATE          1       /* create a new file                    */
#define READ            2       /* read data from a file and return it  */
#define WRITE           3       /* write data to a file                 */
#define DELETE          4       /* delete an existing file              */

/* Error codes. */
#define OK              0       /* operation performed correctly        */
#define E_BAD_OPCODE    -1      /* unknown operation requested          */
#define E_BAD_PARAM     -2      /* error in a parameter                 */
#define E_IO            -3      /* disk error or other I/O error        */

/* Definition of the message format. */
struct message {
    long source;                /* sender's identity                    */
    long dest;                  /* receiver's identity                  */
    long opcode;                /* requested operation                  */
    long count;                 /* number of bytes to transfer          */
    long offset;                /* position in file to start I/O        */
    long result;                /* result of the operation              */
    char name[MAX_PATH];        /* name of file being operated on       */
    char data[BUF_SIZE];        /* data to be read or written           */
};
```

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

An Example Client and Server (2)

- A sample server.

```
#include <header.h>
void main(void) {
    struct message ml, m2;                /* incoming and outgoing messages      */
    int r;                                /* result code                         */

    while(TRUE) {                         /* server runs forever                 */
        receive(FILE_SERVER, &ml);        /* block waiting for a message         */
        switch(ml.opcode) {               /* dispatch on type of request         */
            case CREATE:   r = do_create(&ml, &m2); break;
            case READ:     r = do_read(&ml, &m2); break;
            case WRITE:    r = do_write(&ml, &m2); break;
            case DELETE:   r = do_delete(&ml, &m2); break;
            default:       r = E_BAD_OPCODE;
        }
        m2.result = r;                    /* return result to client             */
        send(ml.source, &m2);             /* send reply                          */
    }
}
```


From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

An Example Client and Server (3)

- A client using the server to copy a file.

(a)

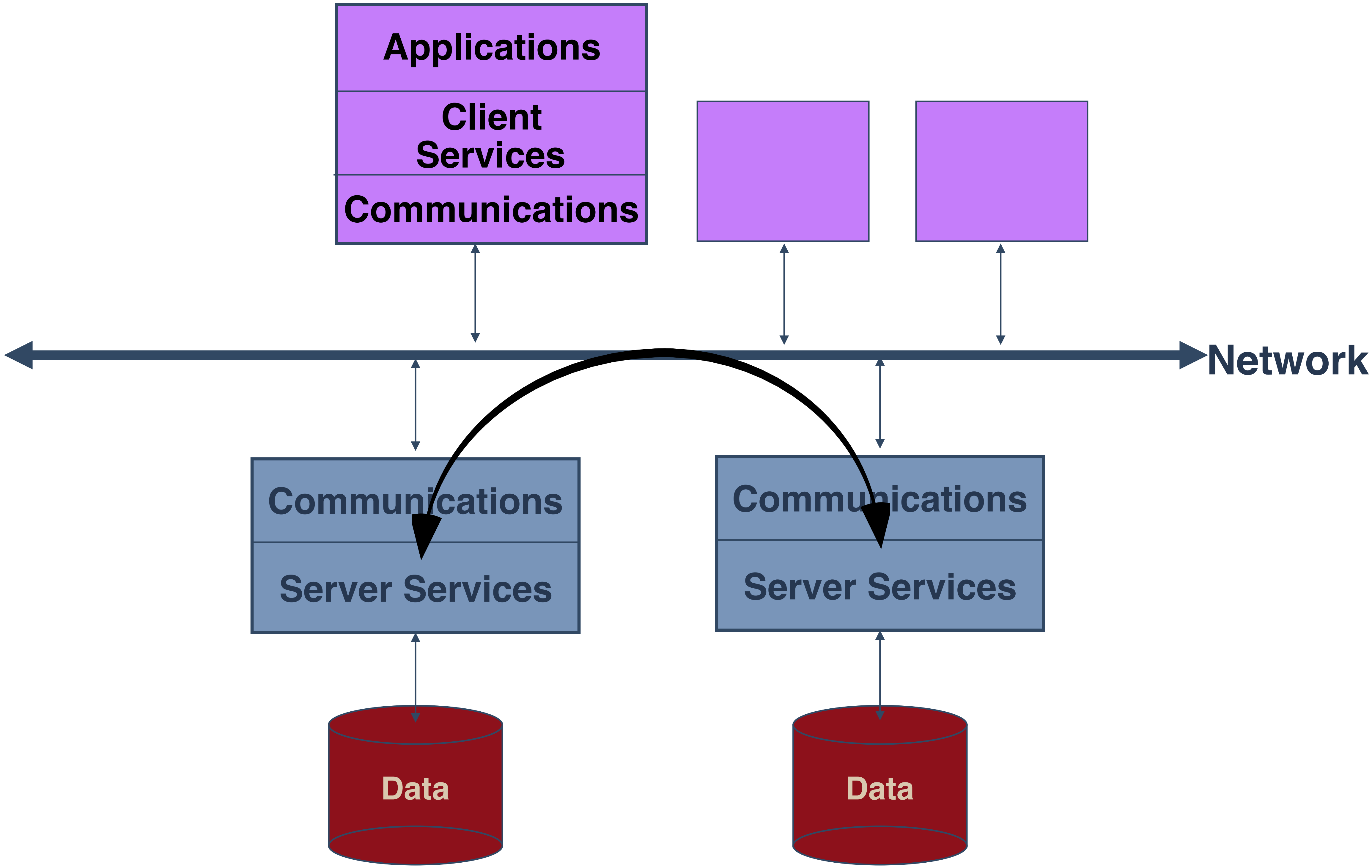
```
#include <header.h>
int copy(char *src, char *dst){              /* procedure to copy file using the server      */
      struct message ml;                     /* message buffer                               */
      long position;                         /* current file position                        */
      long client = 110;                     /* client's address                             */

      initialize( );                         /* prepare for execution                        */
      position = 0;
      do {
            ml.opcode = READ;                /* operation is a read                          */
            ml.offset = position;            /* current position in the file                 */
            ml.count  = BUF_SIZE;                                                        /* how many bytes to read*/
            strcpy(&ml.name, src);           /* copy name of file to be read to message      */
            send(FILESERVER, &ml);           /* send the message to the file server          */
            receive(client, &ml);            /* block waiting for the reply                  */

            /* Write the data just received to the destination file.                         */
            ml.opcode = WRITE;               /* operation is a write                         */
            ml.offset = position;            /* current position in the file                 */
            ml.count  = ml.result;           /* how many bytes to write                      */
            strcpy(&ml.name, dst);           /* copy name of file to be written to buf       */
            send(FILE_SERVER, &ml);          /* send the message to the file server          */
            receive(client, &ml);            /* block waiting for the reply                  */
            position += ml.result;           /* ml.result is number of bytes written         */
      } while( ml.result > 0 );              /* iterate until done                           */
      return(ml.result >= 0 ? OK : ml result); /* return OK or error code                    */
}
```

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms
© Prentice-Hall, Inc. 2002

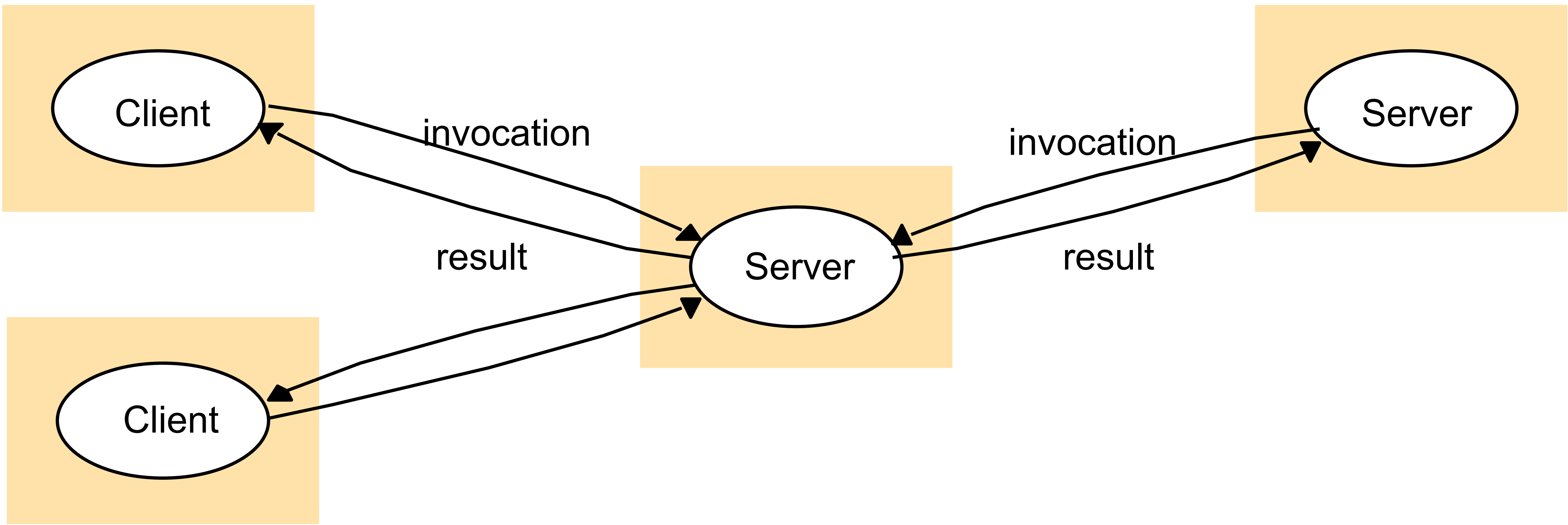
Problems With Multiple-Client/Single Server

- Server forms bottleneck
- Server forms single point of failure
- System scaling difficult

Multiple Clients/Multiple Servers

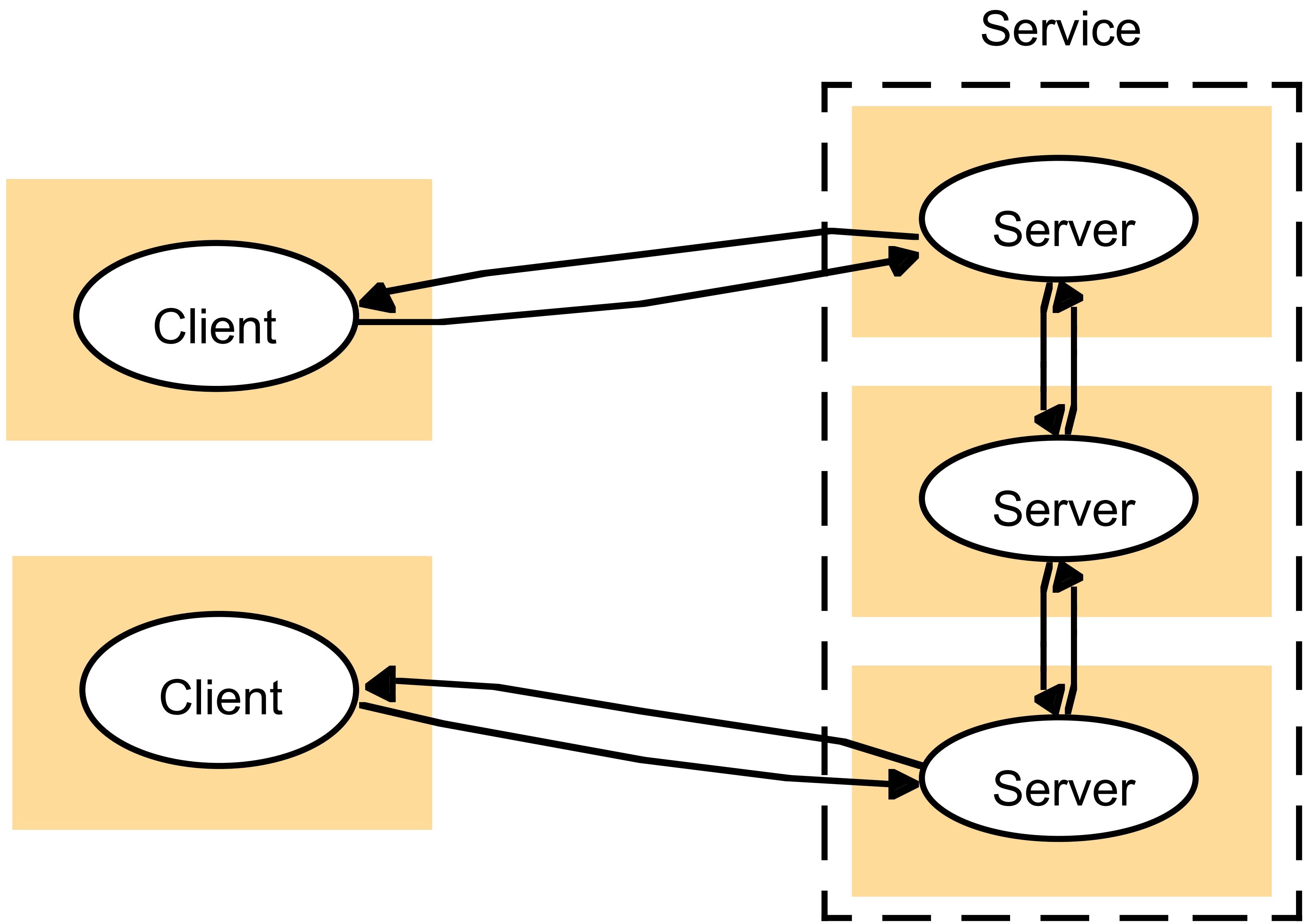


Multiple-Client/Multiple-Server Communication



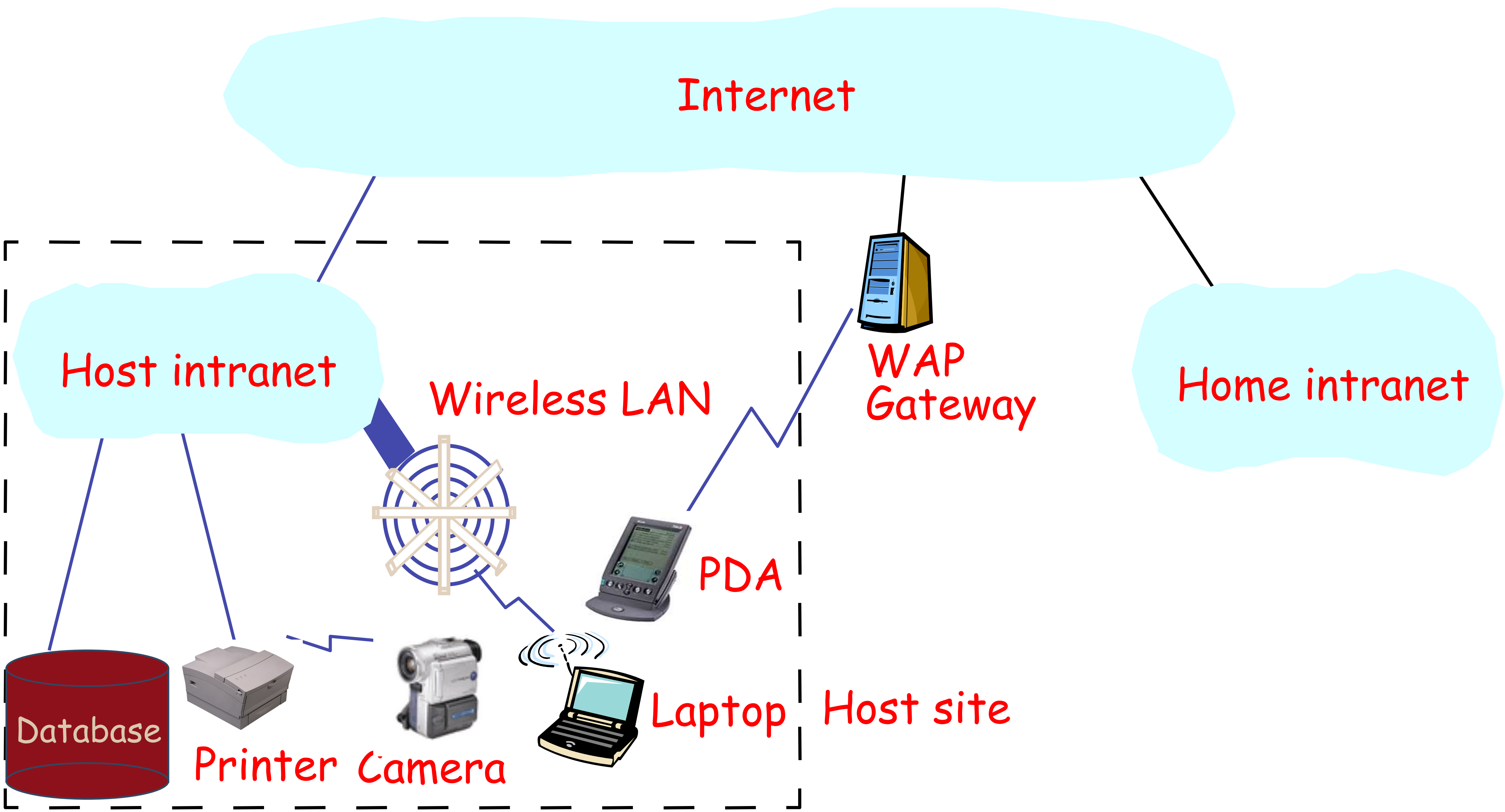
From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

Service Across Multiple Servers

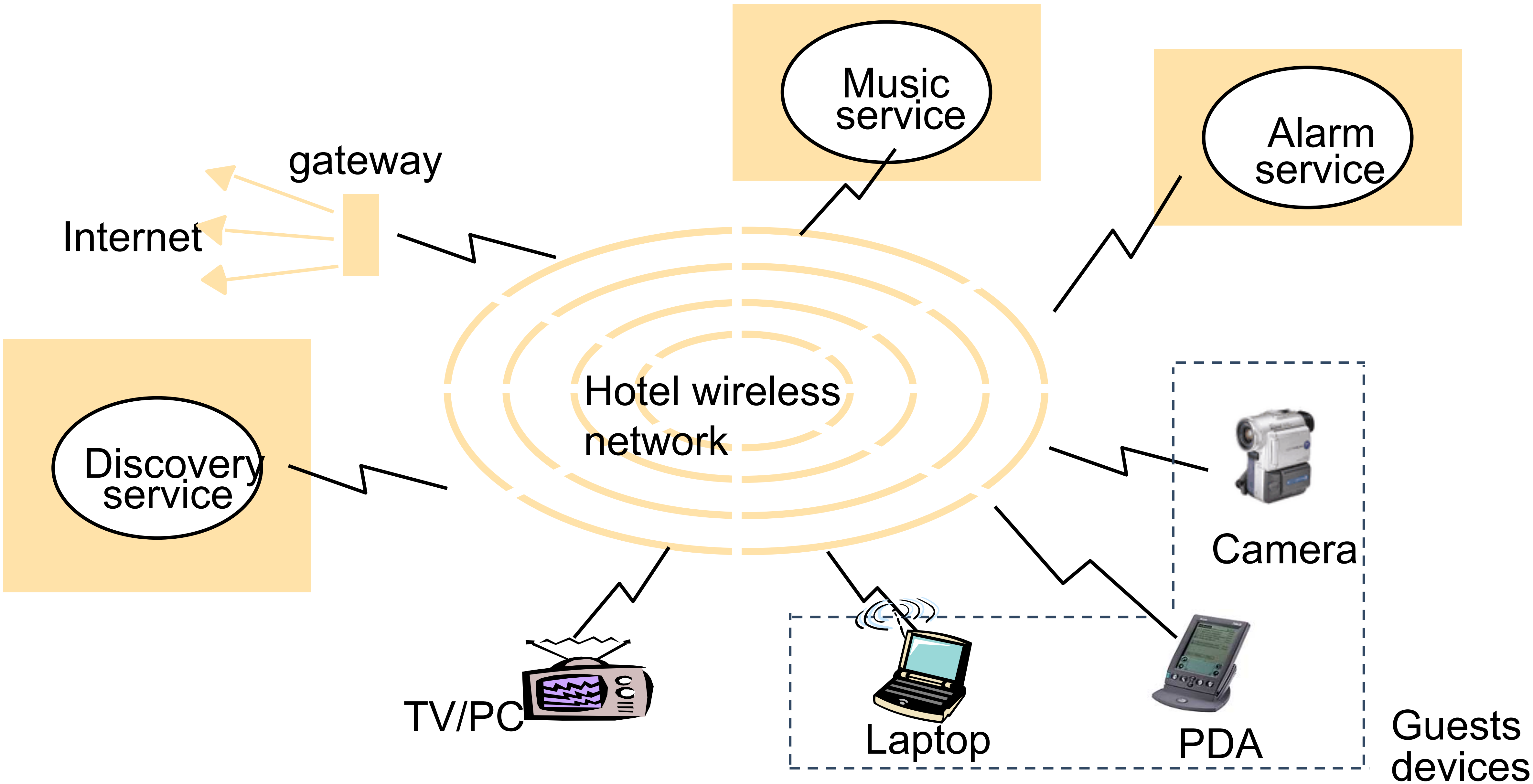


From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

Mobile Computing



Example Mobile Computing Environment

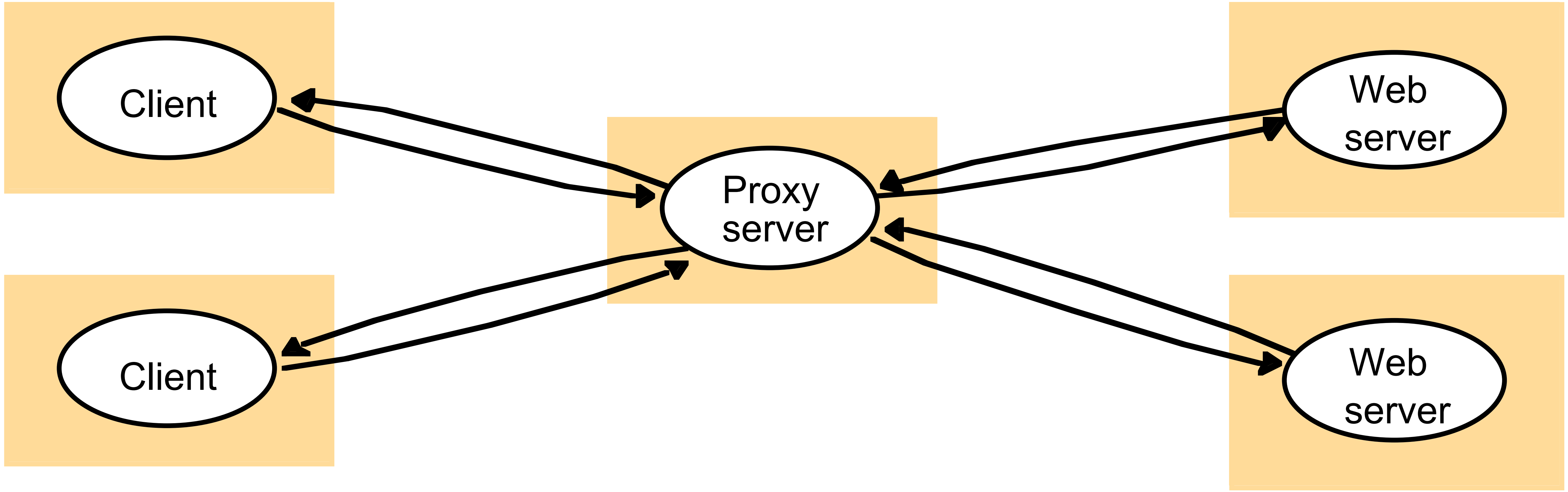


These types of environments are commonly called “spontaneous systems” or “pervasive computing”

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

Multitier Systems

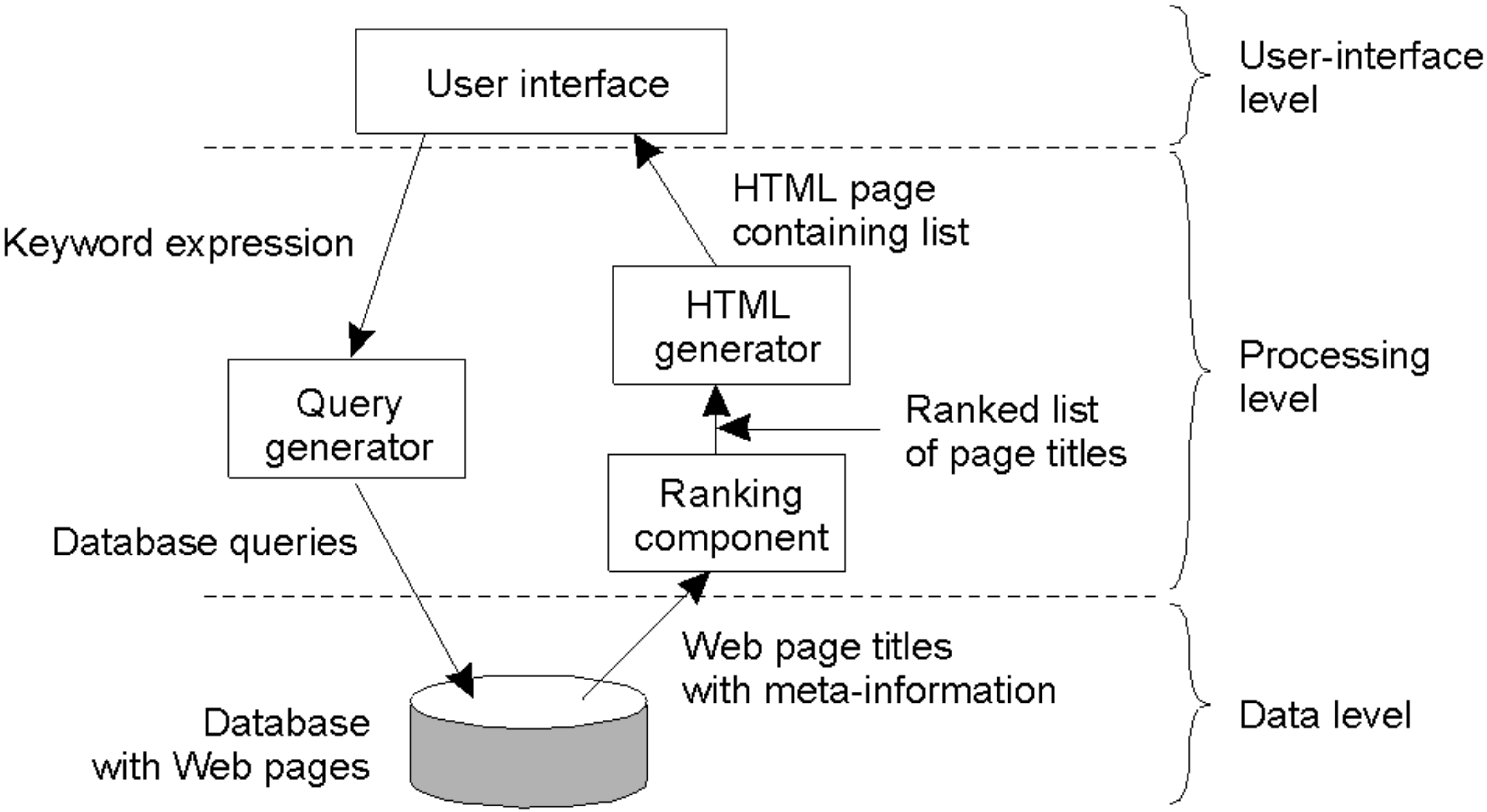
- Servers are clients of other servers
- Example:
  - ➡ Web proxy servers



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

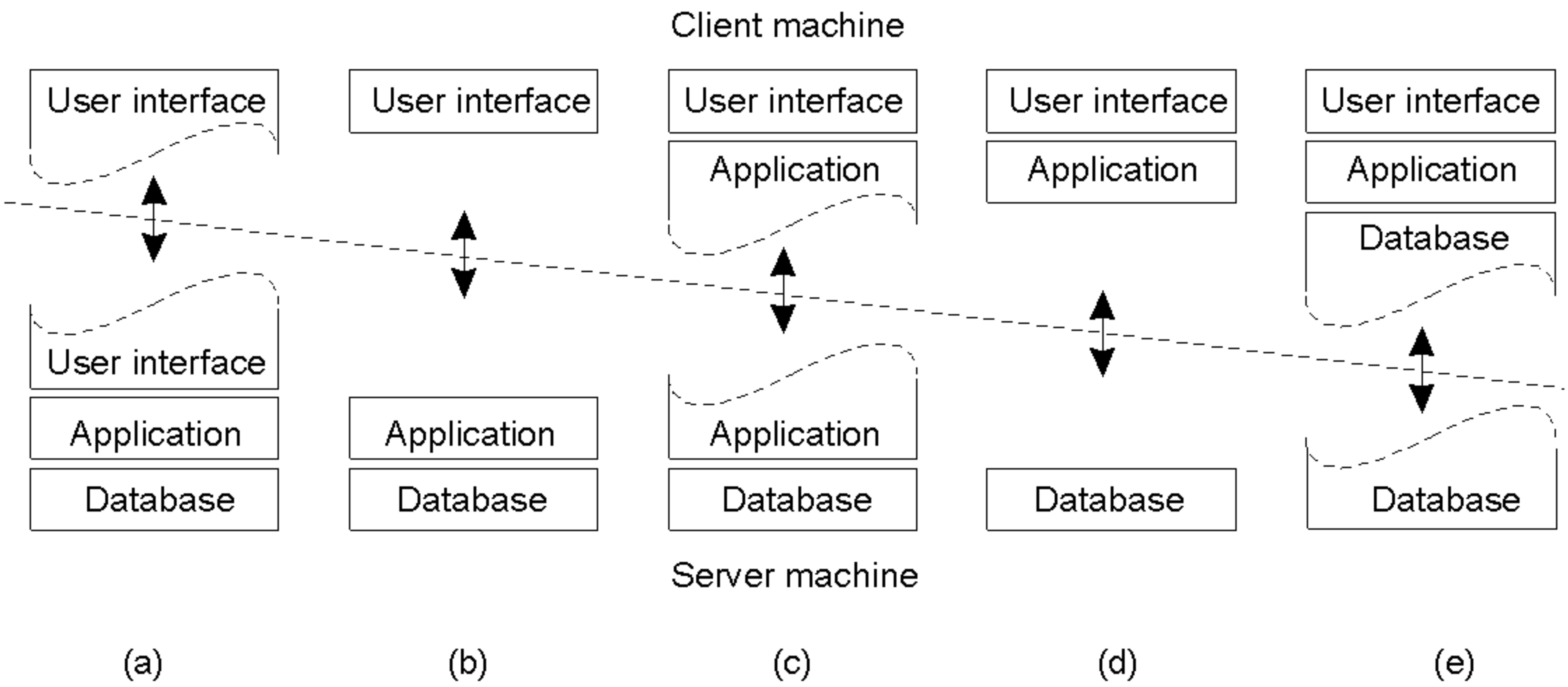
Multitier Systems (2)

- Example: Internet Search Engines



From Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e
© Pearson Education Inc., 2007

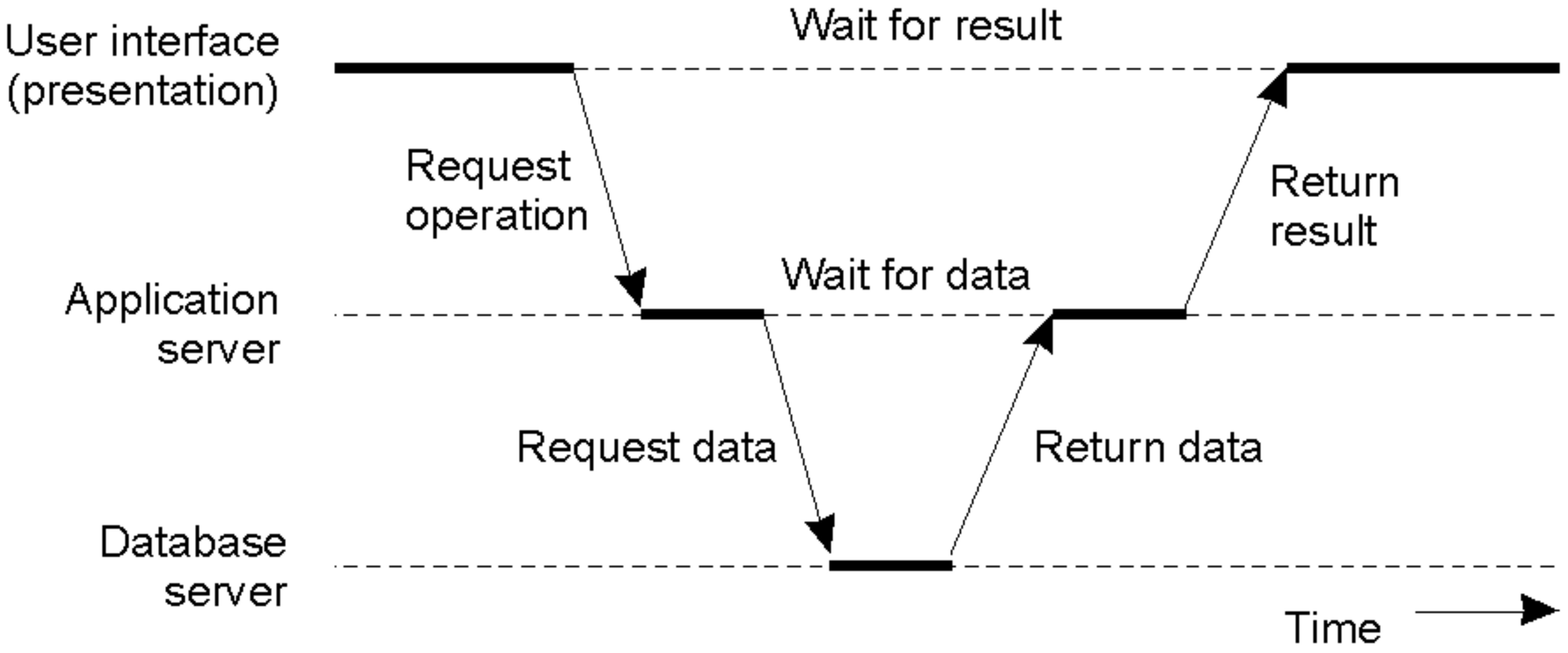
Multitier System Alternatives



From Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e

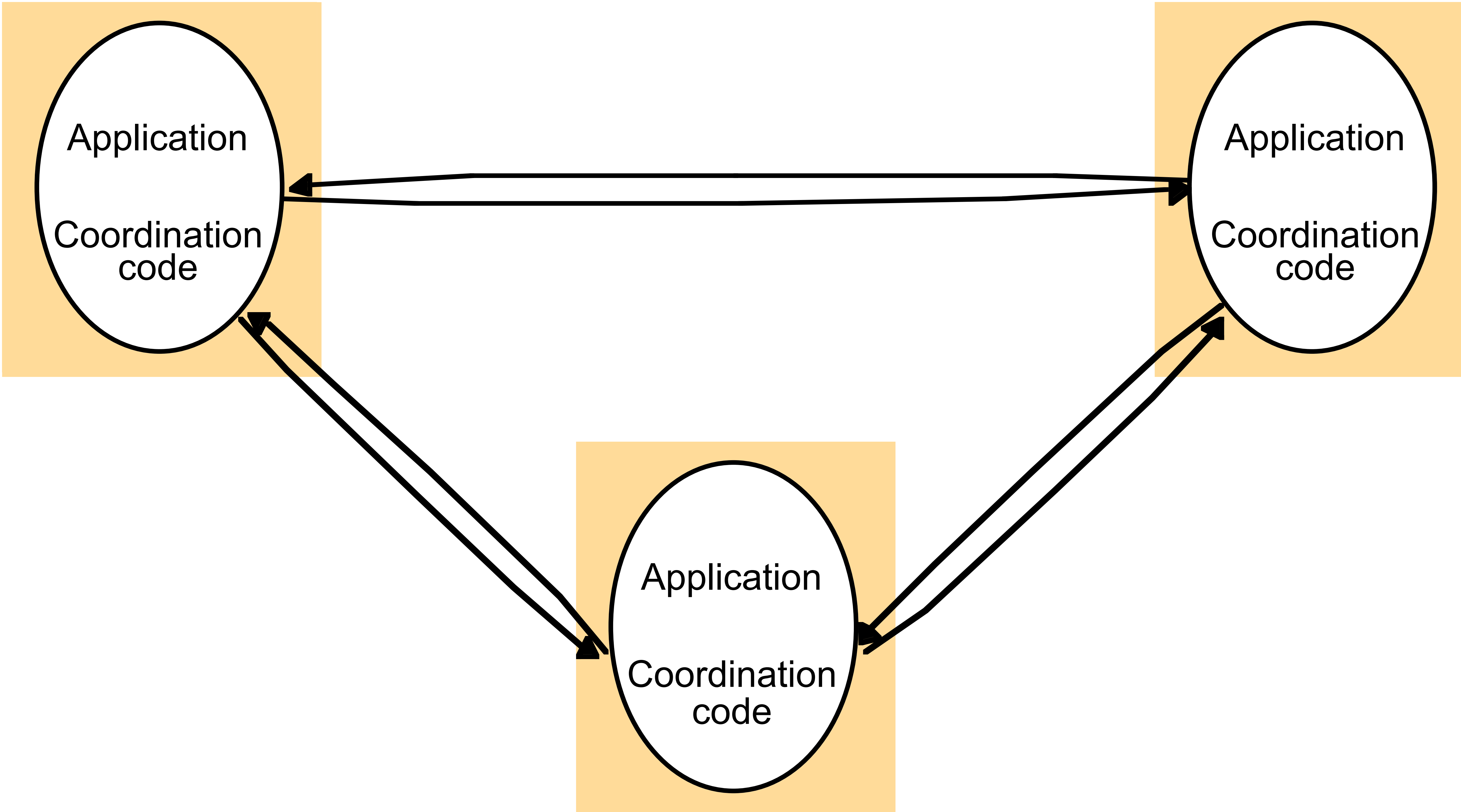
© Pearson Education Inc., 2007

Communication in Multitier Systems



From Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e
© Pearson Education Inc., 2007

Peer-to-Peer Systems



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

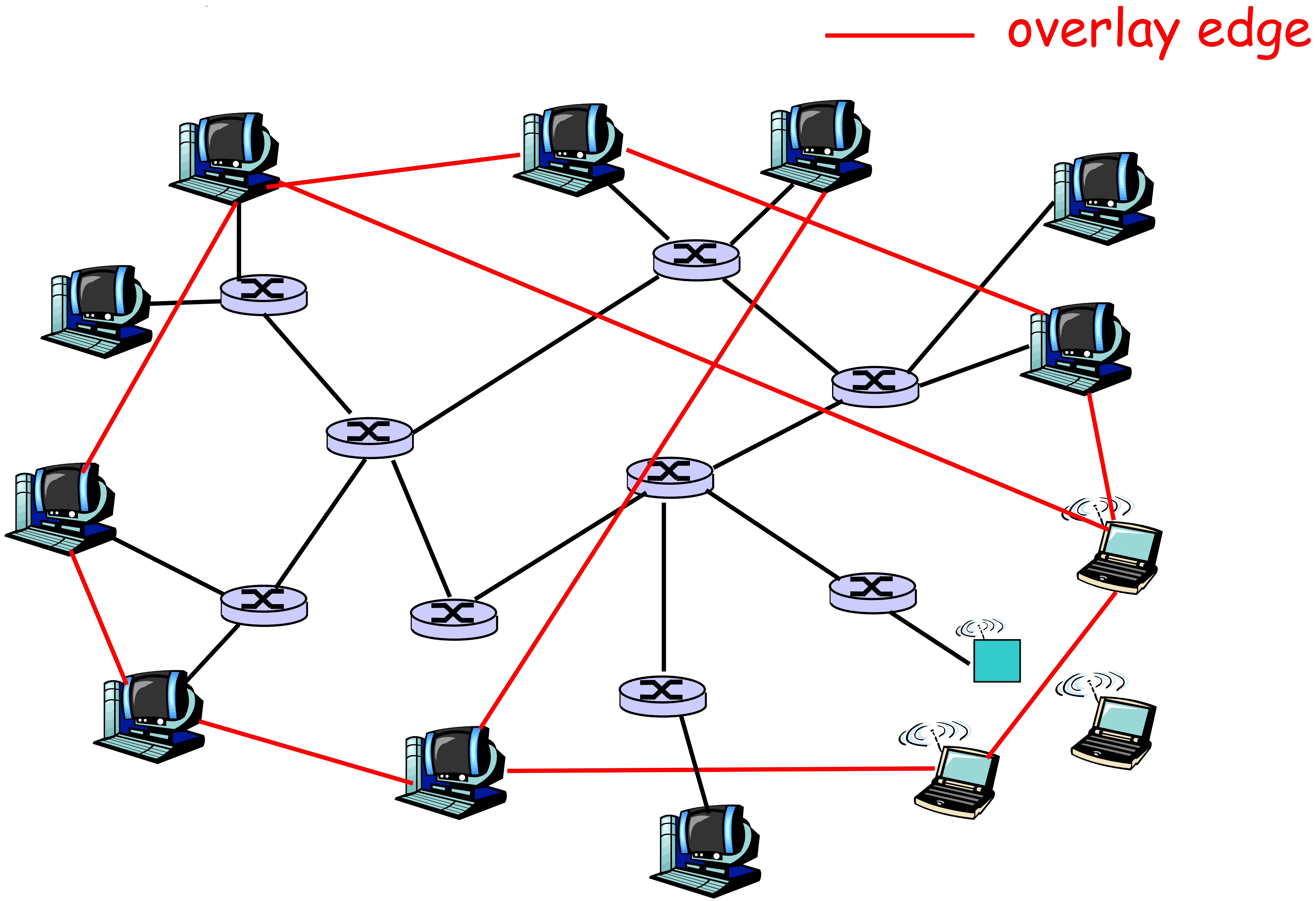
Motivations

- Issues with client/server systems
  - Scalability issues
  - Single point of failure
  - Centralized administration
  - Unused resources at the edge
- P2P systems
  - Each peer can have same functionality (symmetric nodes)
  - Peers can be autonomous and unreliable
  - Very dynamic: peers can join and leave the network at any time
  - Decentralized control, large scale
  - Low-level, simple services
  - File sharing, computation sharing, computation sharing

Potential Benefits of P2P Systems

- Scale up to very large numbers of peers
- Dynamic self-organization
- Load balancing
- Parallel processing
- High availability through massive replication

Overlay Networks

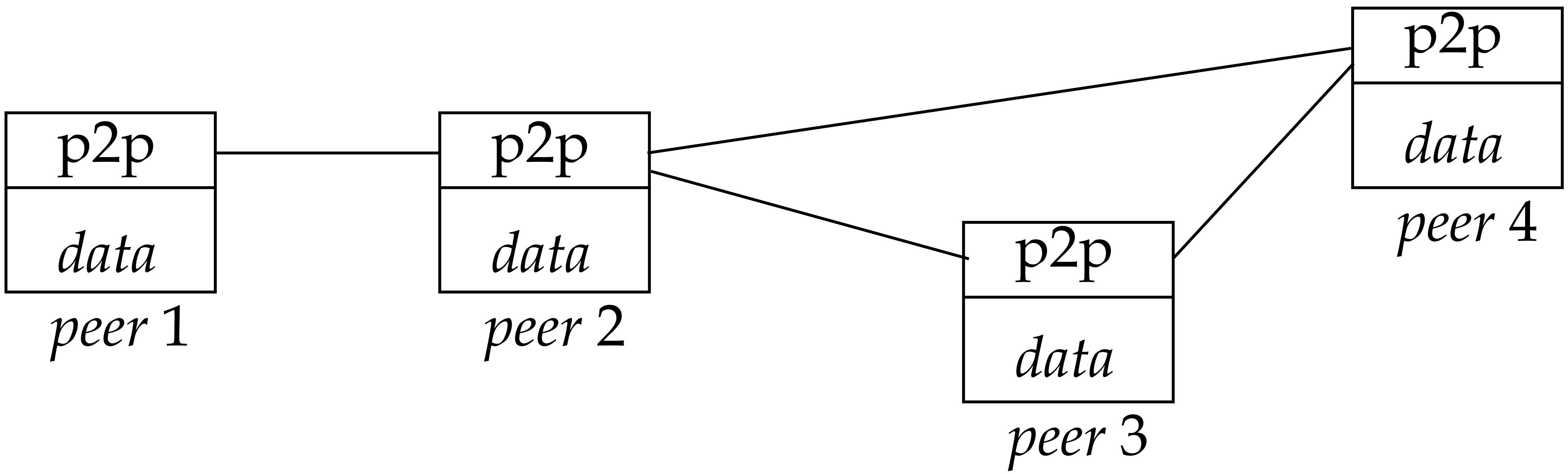


From Ross and Rubenstein, P2P Systems Tutorial

P2P Network Topologies

- Pure P2P systems
  - ➡ Unstructured systems
    - ✦ e.g. Napster, Gnutella, Freenet, Kazaa, BitTorrent
  - ➡ Structured systems (DHT)
    - ✦ e.g. LH* (the earliest form of DHT), CAN, CHORD, Tapestry, Freepastry, Pgrid, Baton
- Super-peer (hybrid) systems
  - ➡ e.g. Edutela, JXTA
- Two issues
  - ➡ Indexing data
  - ➡ Searching data

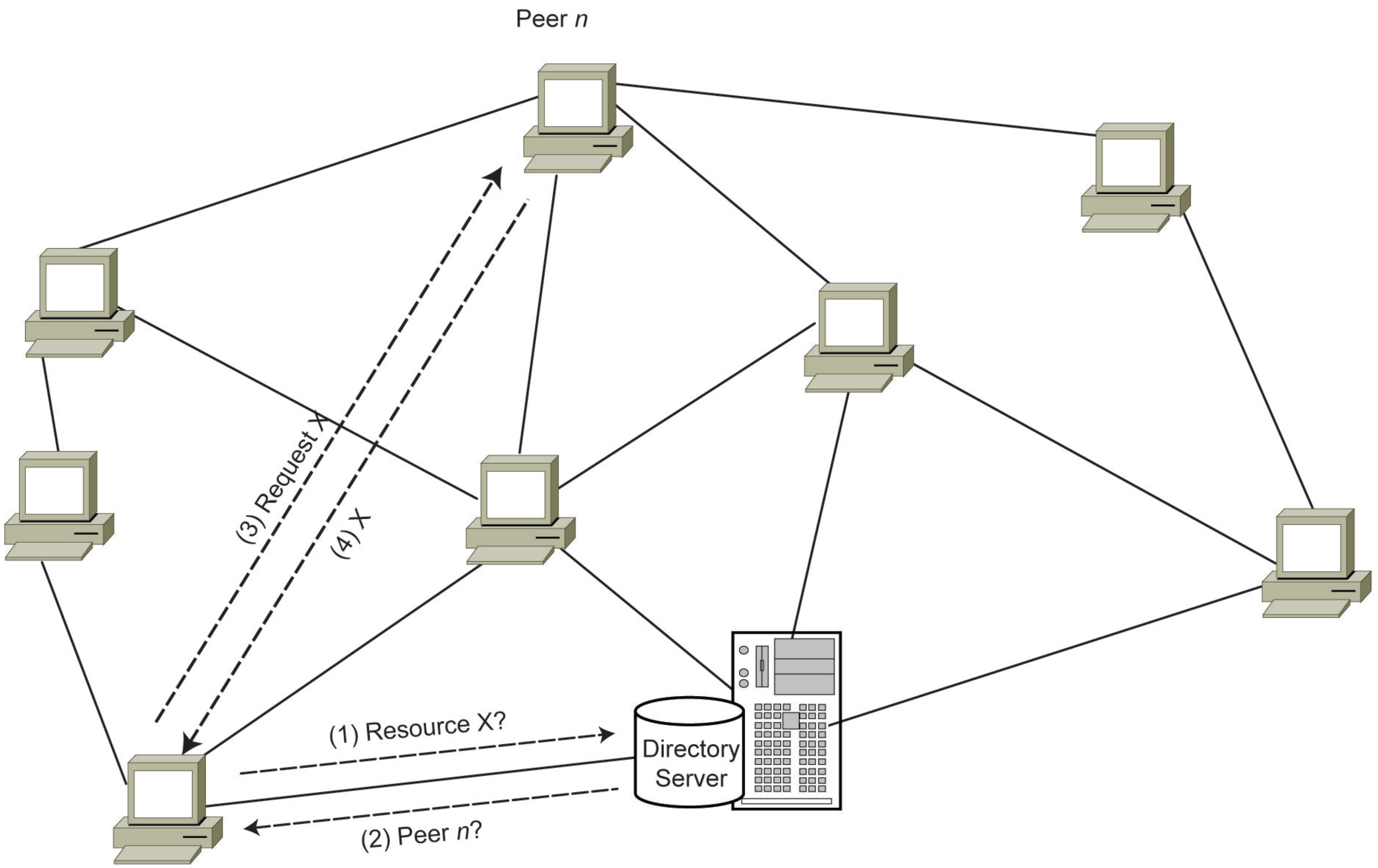
P2P Unstructured Network



- High autonomy (peer only needs to know neighbor to login)
- Searching by
  - ➡ flooding the network: general, may be inefficient
  - ➡ Gossiping between selected peers: robust, efficient
- High-fault tolerance with replication

Search over Centralized Index

1. A peer asks the central index manager for resource
2. The response identifies the peer with the resource
3. The peer is asked for the resource
4. It is transferred



Centralized Index Example – Napster

- Files are kept at individual servers
- List of files are sent to Napster Central Index Server
- Keyword search the Central Index Server for file
- Server replies with the IP addresses of clients who have the file
- You choose one of the clients (e.g., one with the best transfer rate)
- Transfer data from there

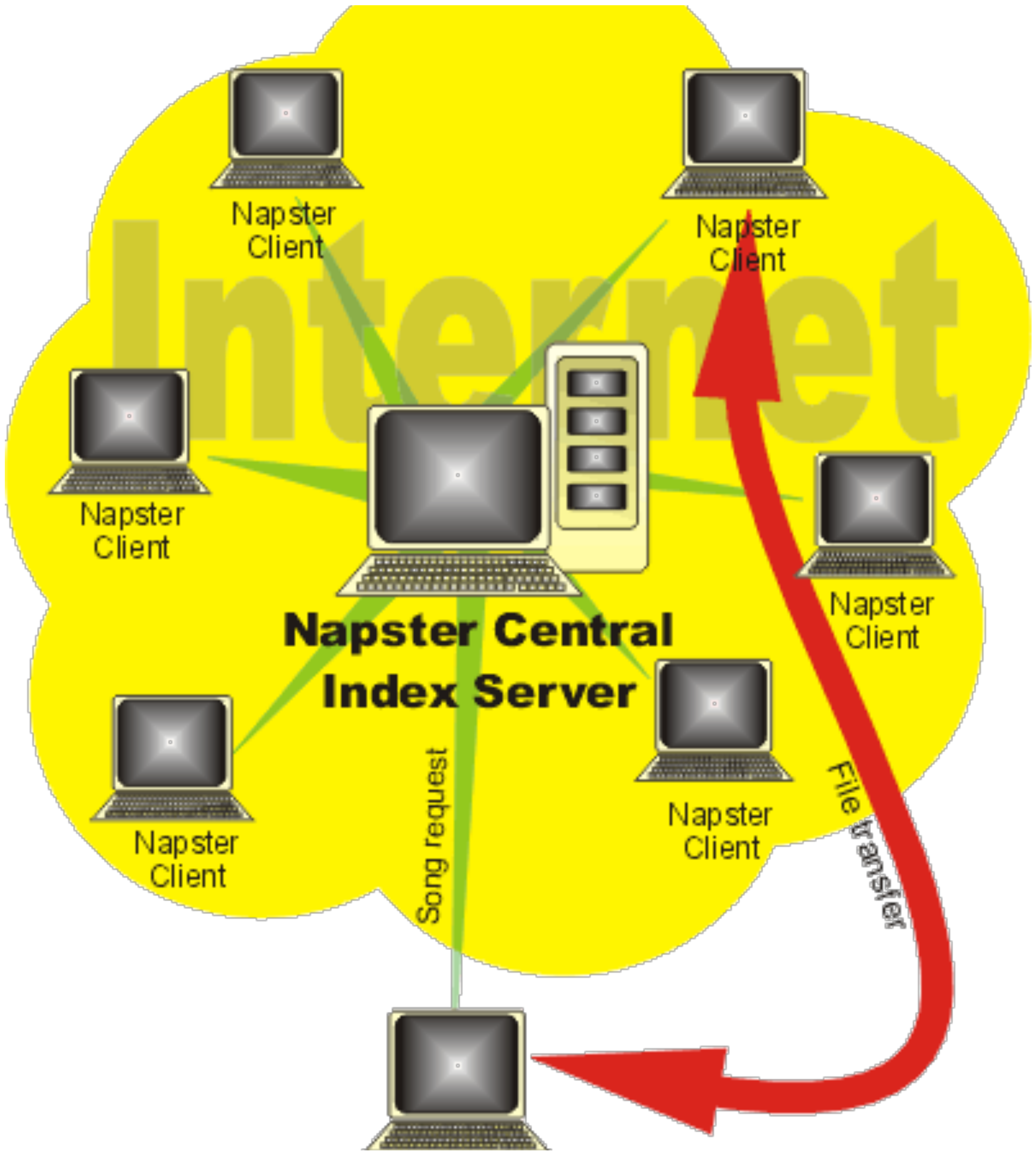
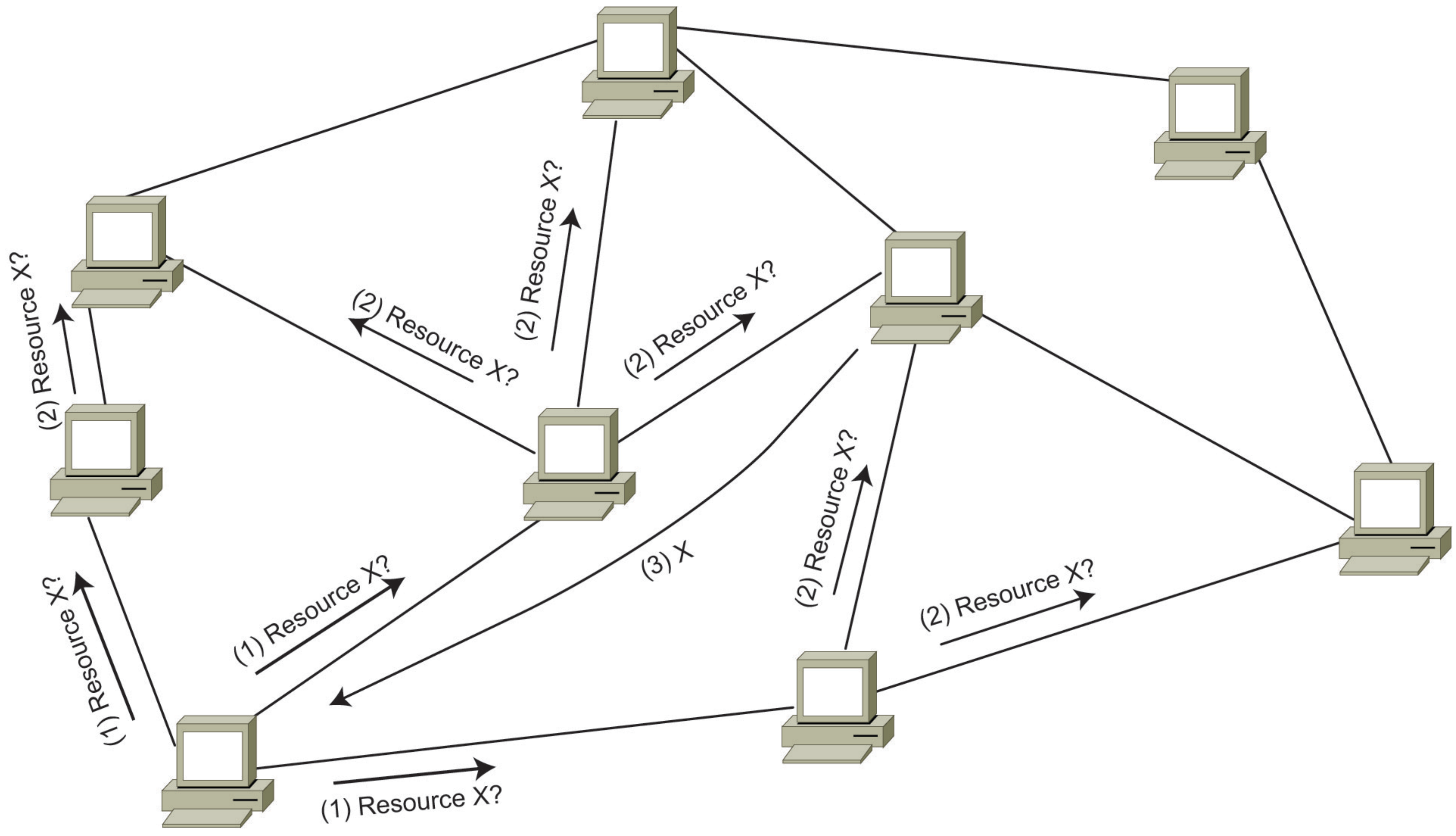


Figure from http://computer.howstuffworks.com/napster.htm

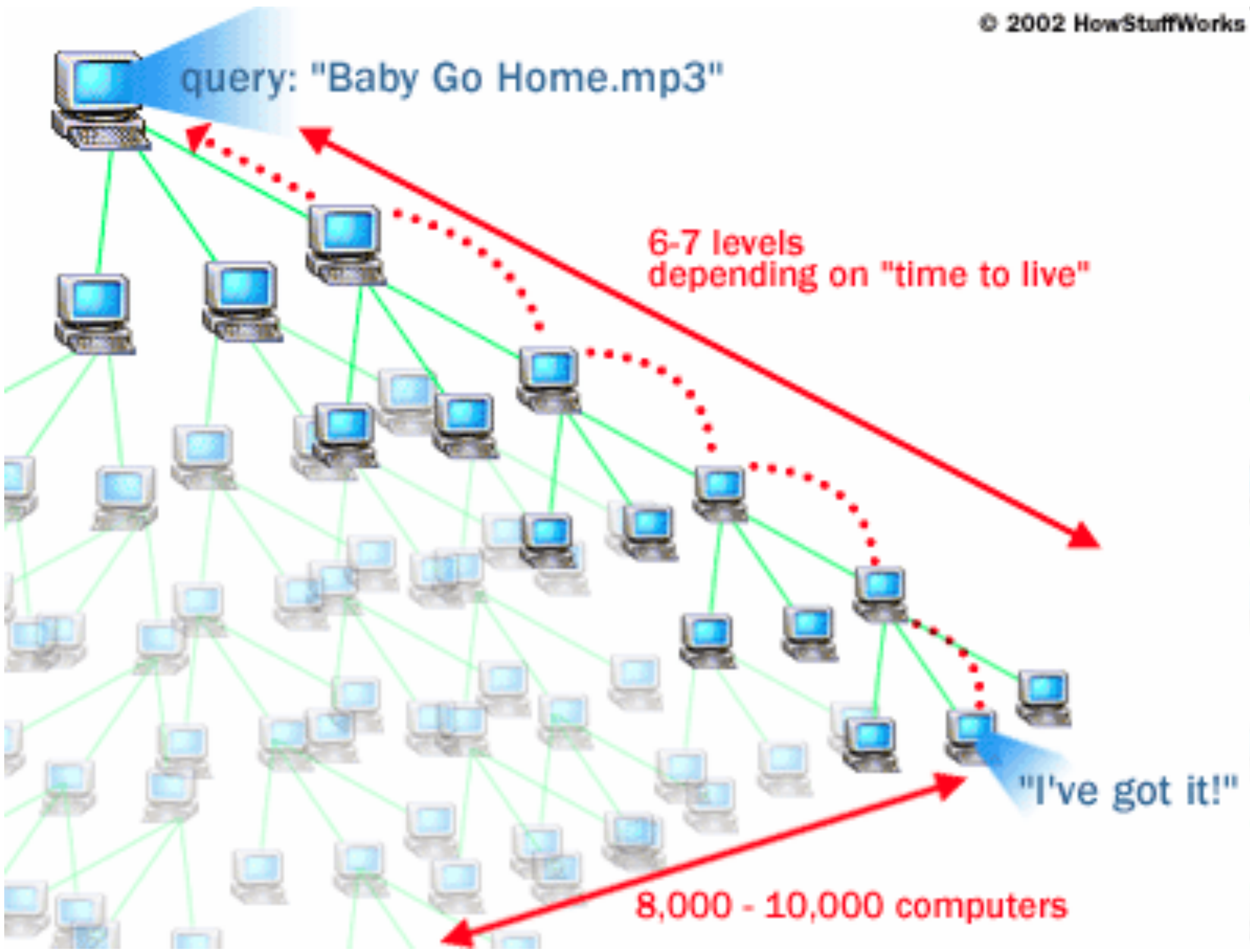
Search over Distributed Index

1. A peer sends the request for resource to all its neighbors
2. Each neighbor propagates to its neighbors if it doesn't have the resource
3. The peer who has the resource responds by sending the resource

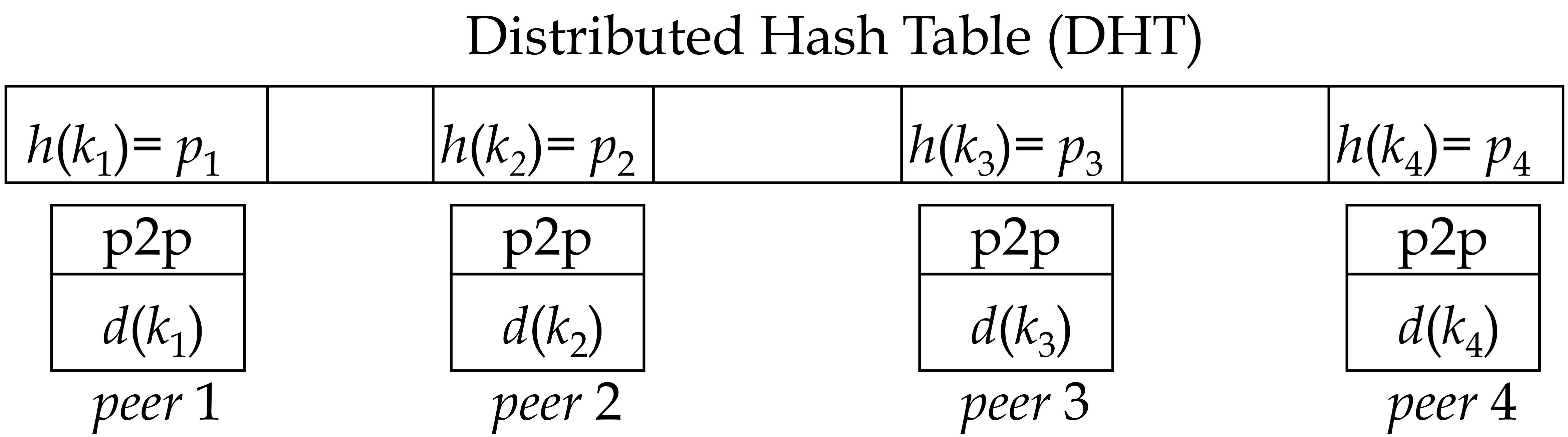


Distributed Index Example – Gnutella

- Flooding
- Controlled by TTL
- Pros
  - ➡ Can reach a large number of computers easily
  - ➡ Always works if you have one connection
  - ➡ No central bottleneck
- Problems
  - ➡ Takes long
  - ➡ Not certain to find
  - ➡ Each client is responsible for routing as well



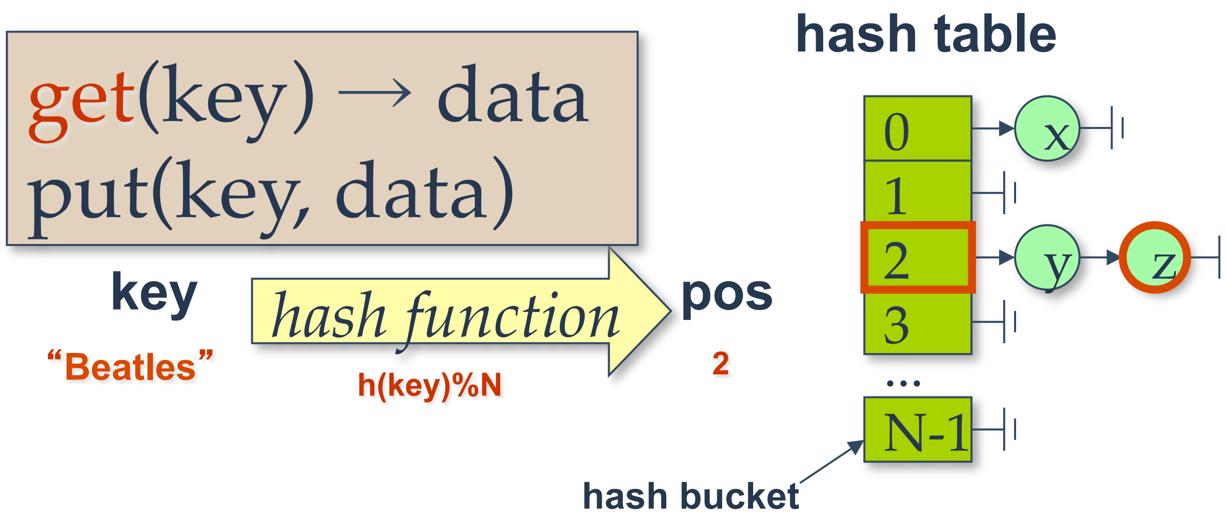
P2P Structured Network

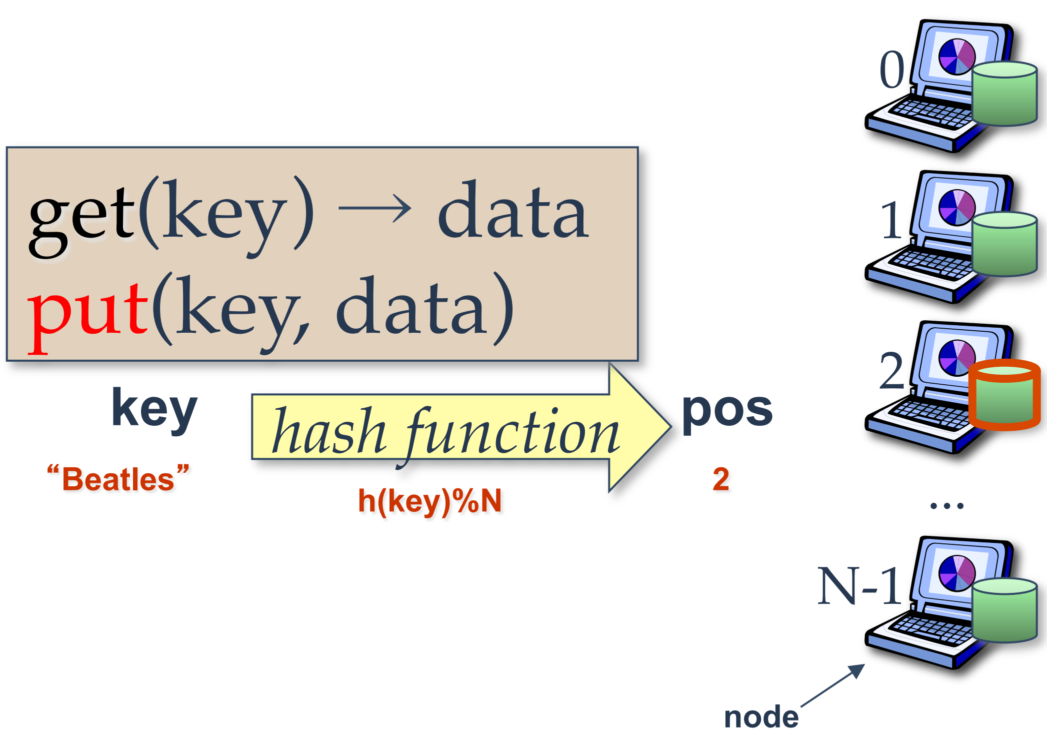


- Simple API with `put(key, data)` and `get(key)`
  - ➡ The key (an object id) is hashed to generate a peer id, which stores the corresponding data
- Efficient exact-match search
  - ➡ $O(log\ n)$ for put(key, data), get(key)
- Limited autonomy since a peer is responsible for a range of keys

Hash Tables

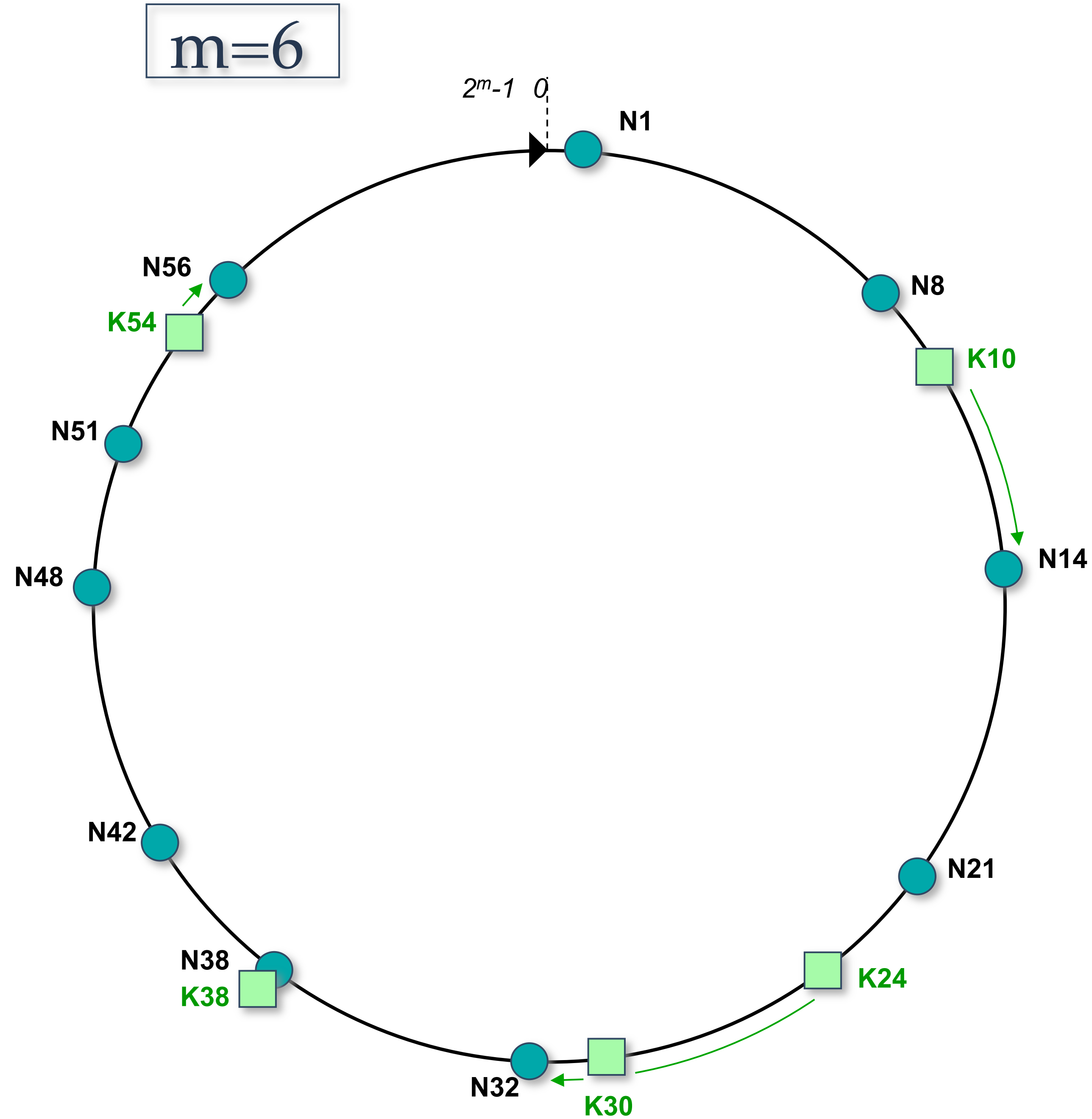
- A hash table associates data with keys
  - ➡ Key is hashed to find bucket in hash table
  - ➡ Each bucket is expected to hold no. items/no. buckets items
- In a Distributed Hash Table (DHT), nodes are the hash buckets
  - ➡ Key is hashed to find responsible peer node
  - ➡ Data and load are balanced across nodes





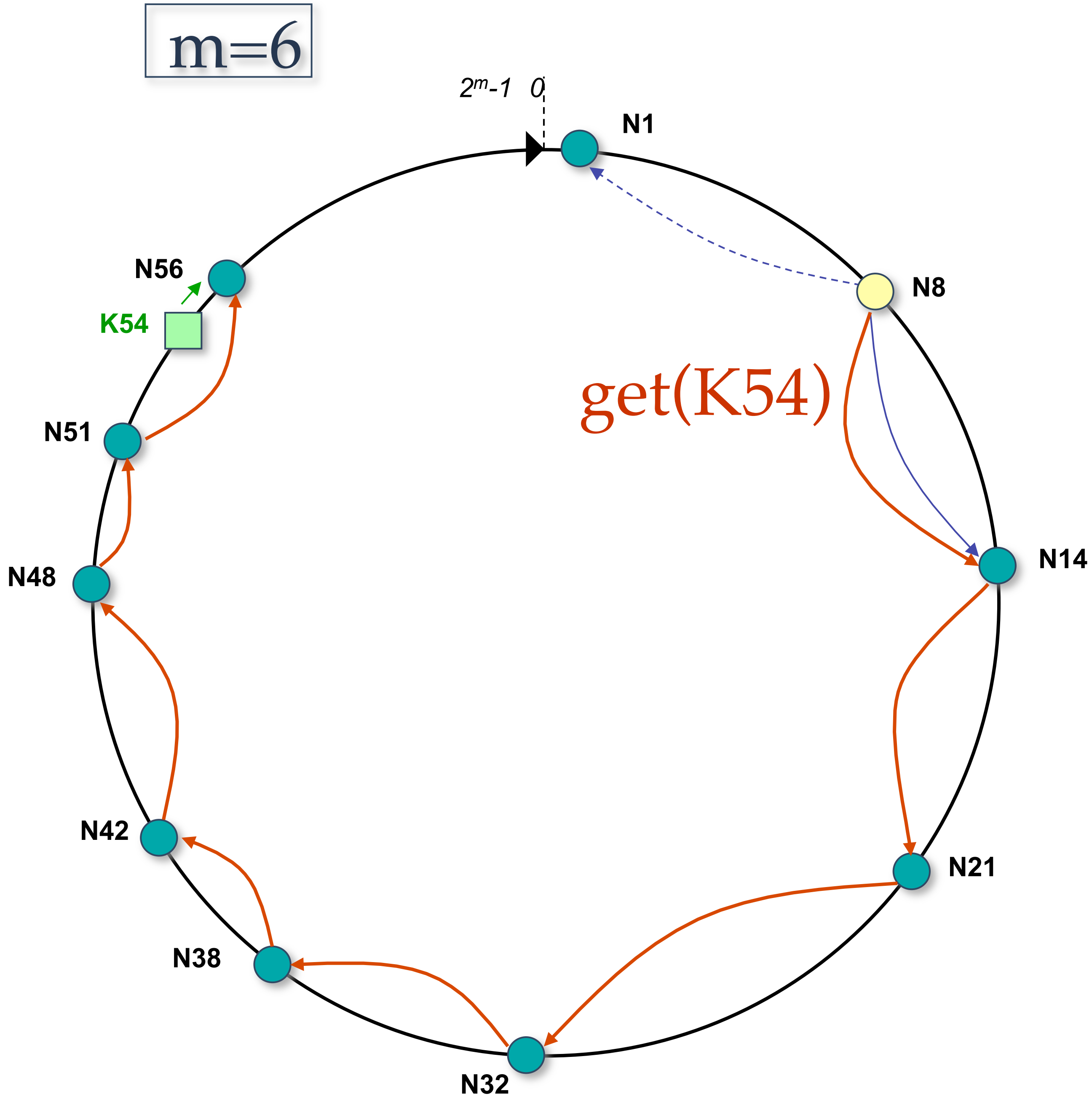
Chord

- Circular *m*-bit ID space for both keys and nodes
- Node ID = SHA-1(IP address)
- Key ID = SHA-1(key)
- A key is mapped to the first node whose ID is equal to or follows the key ID
  - ➡ Each node is responsible for *O(K/N)* keys
  - ➡ *O(K/N)* keys move when a node joins or leaves



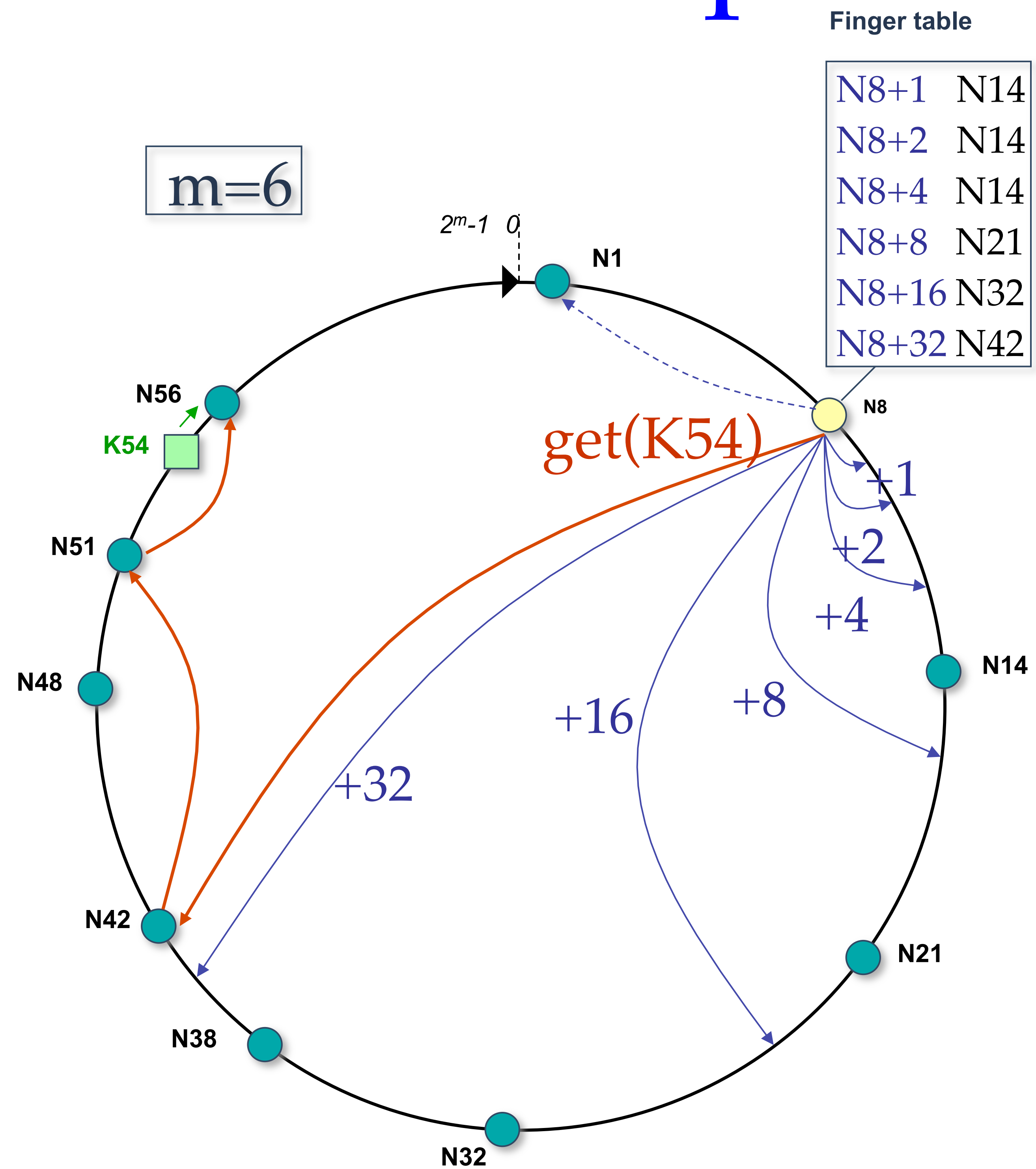
Simple Chord Lookup

- Basic Chord: each node knows only 2 other nodes on the ring
  - ➡ Successor
  - ➡ Predecessor (for ring management)
- Lookup is achieved by forwarding requests around the ring through successor pointers
  - ➡ Requires *O(N)* hops



Improved Chord Lookup

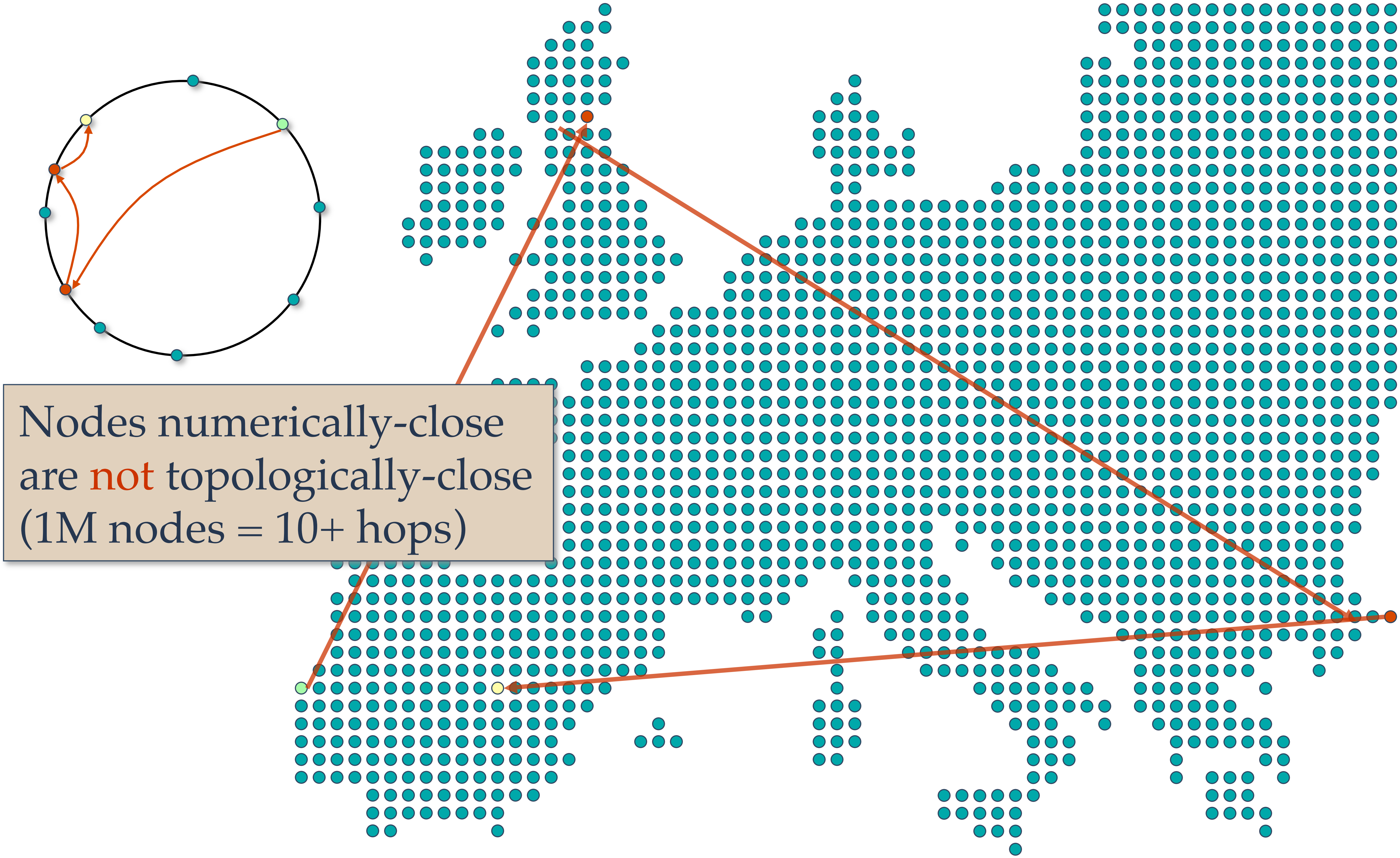
- Each node knows *m* other nodes on the ring
  - ➡ Successors: finger *i* of *n* points to node at $n+2^i$ (or successor)
  - ➡ Predecessor (for ring management)
  - ➡ *O(log N)* state per node
- Lookup is achieved by following closest preceding fingers, then successor
  - ➡ *O(log N)* hops



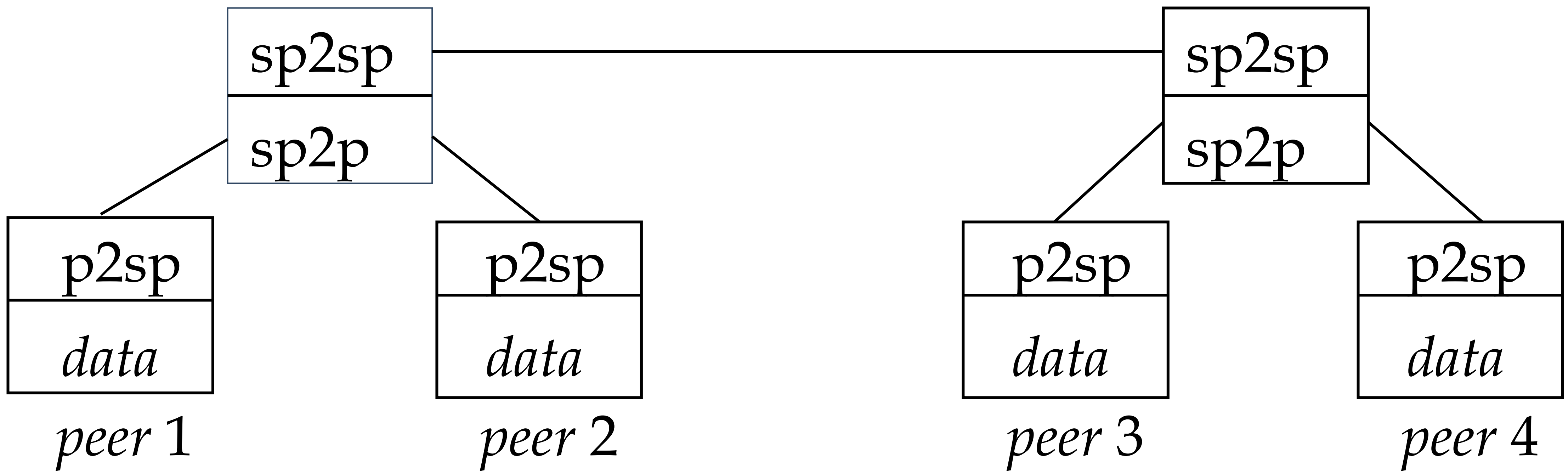
Chord Properties

- In a system with N nodes and K keys, with high probability
  - ➡ each node received K/N keys
  - ➡ each node maintains information about $O(\log N)$ other nodes
  - ➡ lookups resolved with $O(\log N)$ hops
- No delivery guarantees
- No consistency among replicas
- Hops have poor network locality

Chord and Network Topology



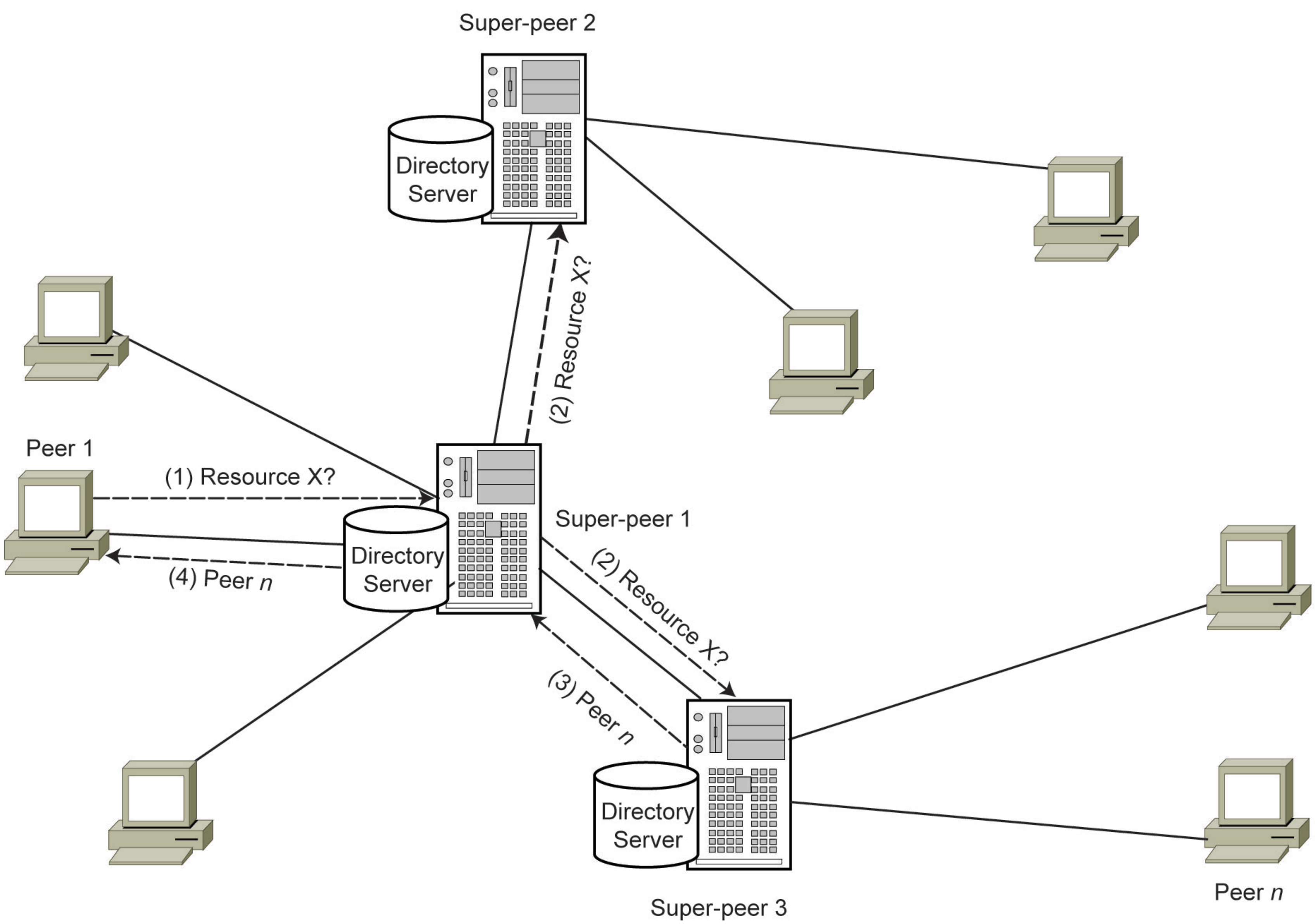
Super-peer Network



- Super-peers can perform complex functions (meta-data management, indexing, access control, etc.)
  - ➡ Efficiency and QoS
  - ➡ Restricted autonomy
  - ➡ SP = single point of failure ➔ use several super-peers

Search over a Super-peer System

1. A peer sends the request for resource to all its super-peer
2. The super-peer sends the request to other super-peers if necessary
3. The super-peer one of whose peers has the resource responds by indicating that peer
4. The super-peer notifies the original peer



Super-Peer System Example – KaZaa

- Unstructured super-peer system
  - ➡ Possible to do super-peers (i.e., hierarchical search) over DHTs as well
- List of potential super-peers included within software download
- New peer goes through list until it finds operational super-peer
  - ➡ Connects, obtains more up-to-date list, with 200 entries
  - ➡ Node then pings 5 nodes on list and connects with the one
- If super-peer goes down, node obtains updated list and chooses new super-peer

P2P Systems Comparison

Requirements	Unstructured	DHT	Super-peer
Autonomy	high	low	avg
Query exp.	high	low	high
Efficiency	low	high	high
QoS	low	high	high
Fault-tolerance	high	high	low
Security	low	low	high