CompSci 131

Parallel and Distributed Systems

Prof. A. Veidenbaum

Today's topics

- **Last Lecture Summary**
- **Middleware organization**
- **System Architecture**

- **Reading assignment:**
  - Today's: Sec. 2.2-2.3
  - Next time: Sec. 2.4 (on your own), 3.1
    - » Also start reading this pthreads tutorial
    - » https://computing.llnl.gov/tutorials/pthreads/
  - Complete the assignment ***before*** next class

Review of last lecture

- **Architectural styles in DS**
  - **Layered**
  - **Object-based**
  - **RESTful**
  - **Event-based**
  - **Publish/subscribe**

Middleware organization

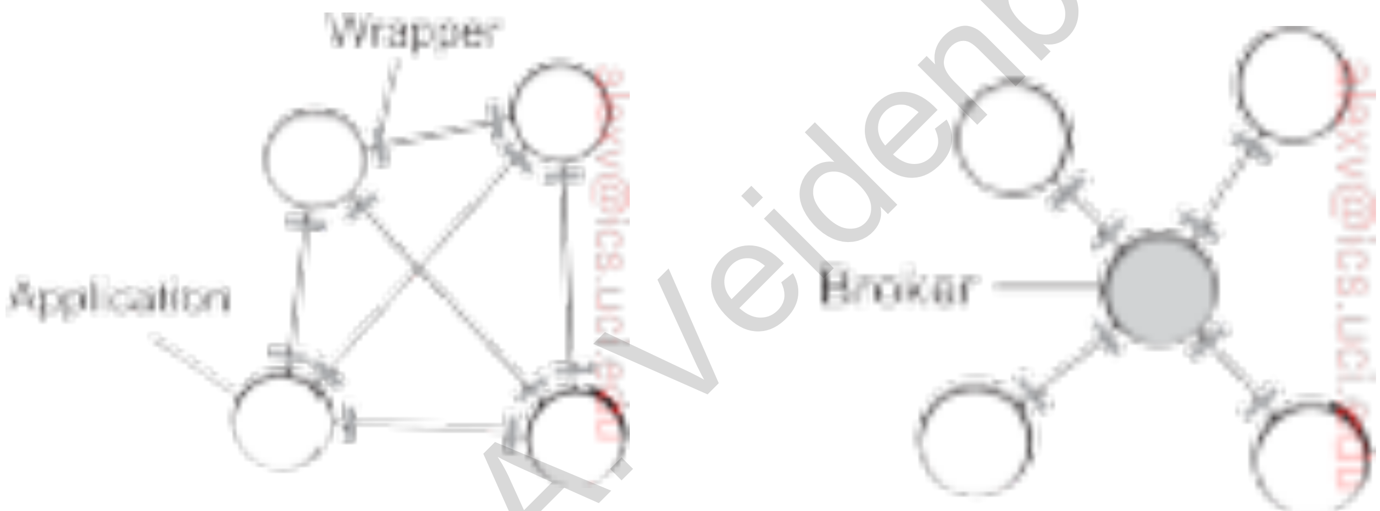
- **It is distinct and separate from *software* architecture**
- **There are two important types of *design patterns* used in middleware organization:**
  - **Wrappers or adapters**
  - **Interceptors**
- **Both target the same goal – DS openness**
- **Arguably, the ultimate openness is achieved when the middleware can be composed at runtime**
  - **Modifiable middleware**

Wrappers/Adapters

- **Solve the problem of expanding a component interface**
  - **The old component and its interface may remain unchanged for legacy or other reasons**
  - **A new interface provided by a wrapper may be translated to the old component interface**
- **Have their origin in OO programming**
  - **More complex in DS**
- **Extensibility problem – unique application to application interfaces were used to achieve it**
  - **N applications would require O(N^2) unique wrappers**
- **Special middleware may be used for such a reduction**
  - **For instance use a *broker,* such a message broker**

Wrappers/Adapters and Brokers

- **A broker knows how to handle any message and sends it to an appropriate component**
  - **Uses 1-to-N and N-to-1 interfaces, e.g. O(N)**



Interceptors

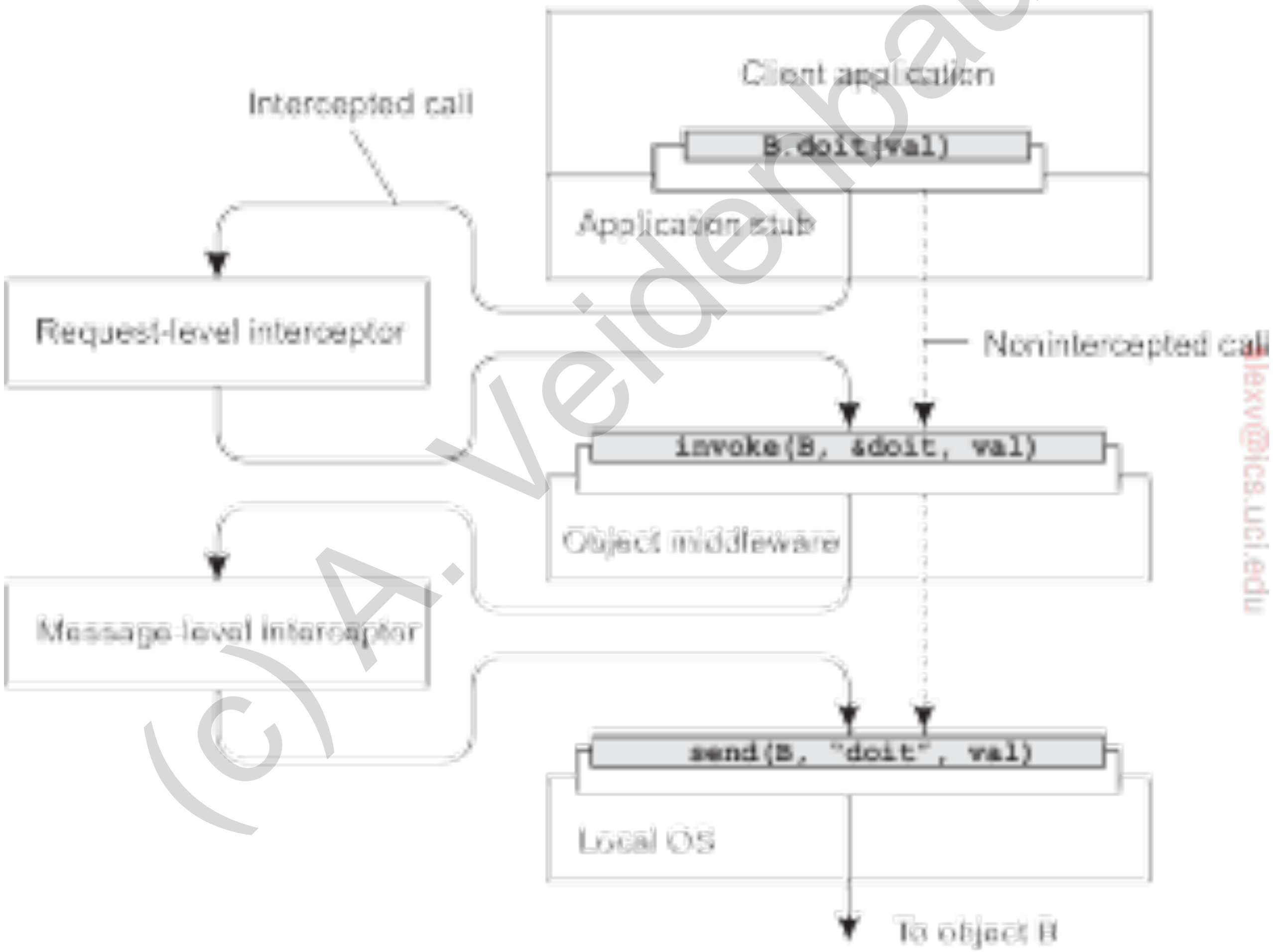
- A *software* construct that can break normal control flow and allow an app-specific code to be executed
  – E.g. it adapts middleware to the needs of an application
  – This makes middleware more open

- Let us consider an object-based DS in which an object A can call a method in object B
  – The objects reside in different nodes

- This is done as follows
  1. Obj. A is offered a local interface equivalent to B's
  2. The call by A is transformed into a generic obj. invocation
     » The general obj. invocation interface is offered locally by the middleware
  3. The generic obj. invocation is transformed into a message and sent to B

Interceptors

- **The example shows two interceptors**
  1. **A message interceptor we discussed on previous slide**
  2. **A request interceptor that can be used for replicated obj.**



Modifiable middleware

- **Wrappers and interceptors are a means to adapt to continuous changes in DSs.**
  - **Failures, delays, replication, system scaling, etc.**
- **Middleware is made to adapt to such changes**
  - **Applications are blissfully unaware of any**
- **Scalability, in particular, makes it very desirable to modify middleware without bringing the DS down**
  - **Adaptive or modifiable middleware does this**
    - **Replacing a component at runtime is a good example**
- **One approach is to *dynamically construct* middleware from components**
  - **As opposed to statically configuring it at design time**

Modifiable middleware

- Dynamic configuration is complex
  - Need support for late binding
  - Need to know the effect of component replacement on all other affected components

- A minimum requirement is to provide dynamic loading/unloading of components

- This is complex and is the subject of current research

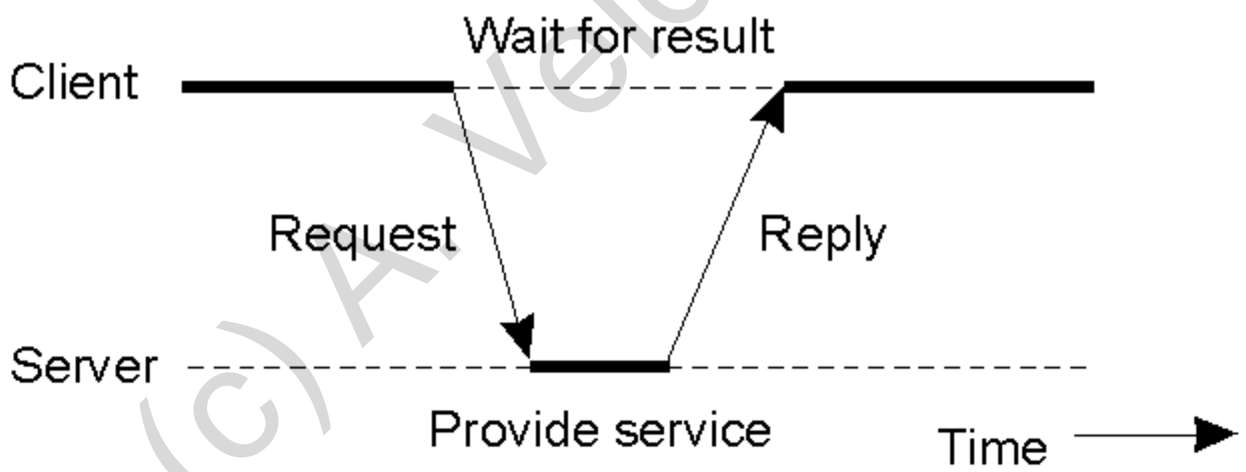
System Architectures

- **Recall that *A system* architecture defines software instantiated on a real system**
  - **After final configuration choices were made**
  - **Component placement is decided**

- **There are two main types**
  - **Centralized**
    - » **Client-server, with clients sending requests**
  - **Distributed**
    - » **Peer-to-Peer**

Client/Server Architectures

- **A good model for interaction in a distributed systems**
  - **Servers provide a service abstraction**
  - **Clients request services**
- **Here is a typical interaction between a client and a server**
  - **Or a request-reply behavior**



- **Can use a simple connectionless communication**
  - **If the network is reliable or if message loss acceptable**
- **Or use TCP/IP otherwise**
  - **But it is less efficient**
- **Server will interpret and do the requested work**

A Client/Server Example

- 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       */
};
```

An Example of a 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                         */
    }
}
```

Client File Copy Example

```
#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                       */
}
```

Multitiered architectures

- Recall the three levels we discussed
  - User interface level
    - » Deals with input, output functions like display, getting input
  - Processing level
    - » This is the application itself
  - Data level
    - » Stores the state or data and defines primitive operations on it
- Many different implementations are possible
  - Along the three levels above
- Let us start with a (physically) 2-tiered architecture
  - Client system and server system

Client-Server Architectures

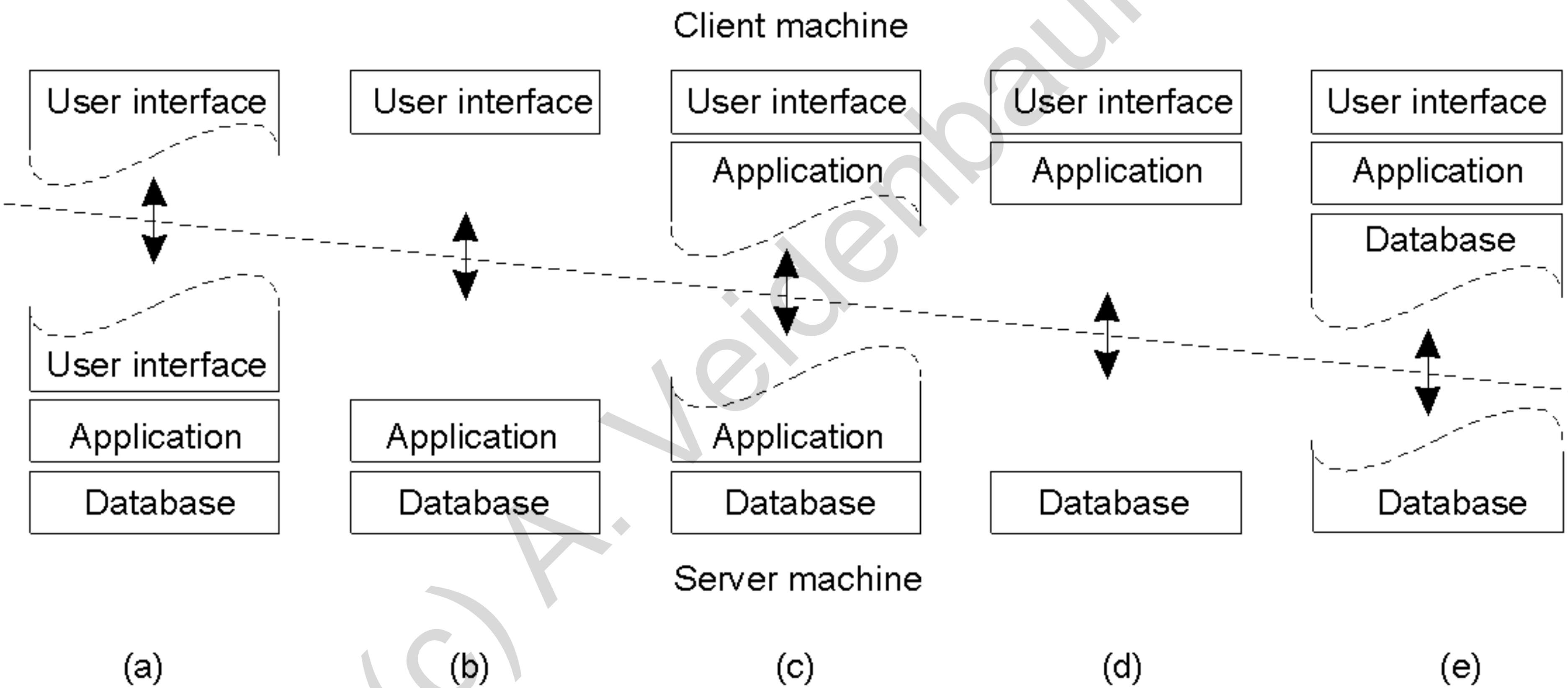
- **Simplest:**
  - **Client implements the user interface level**
  - **Server implements the rest**

- **What is the problem with this system?**

- **Many other ways possible**
  - **Can actually distribute programs in a level/layer**
  - **Two-tiered architecture**

Multitiered Architectures

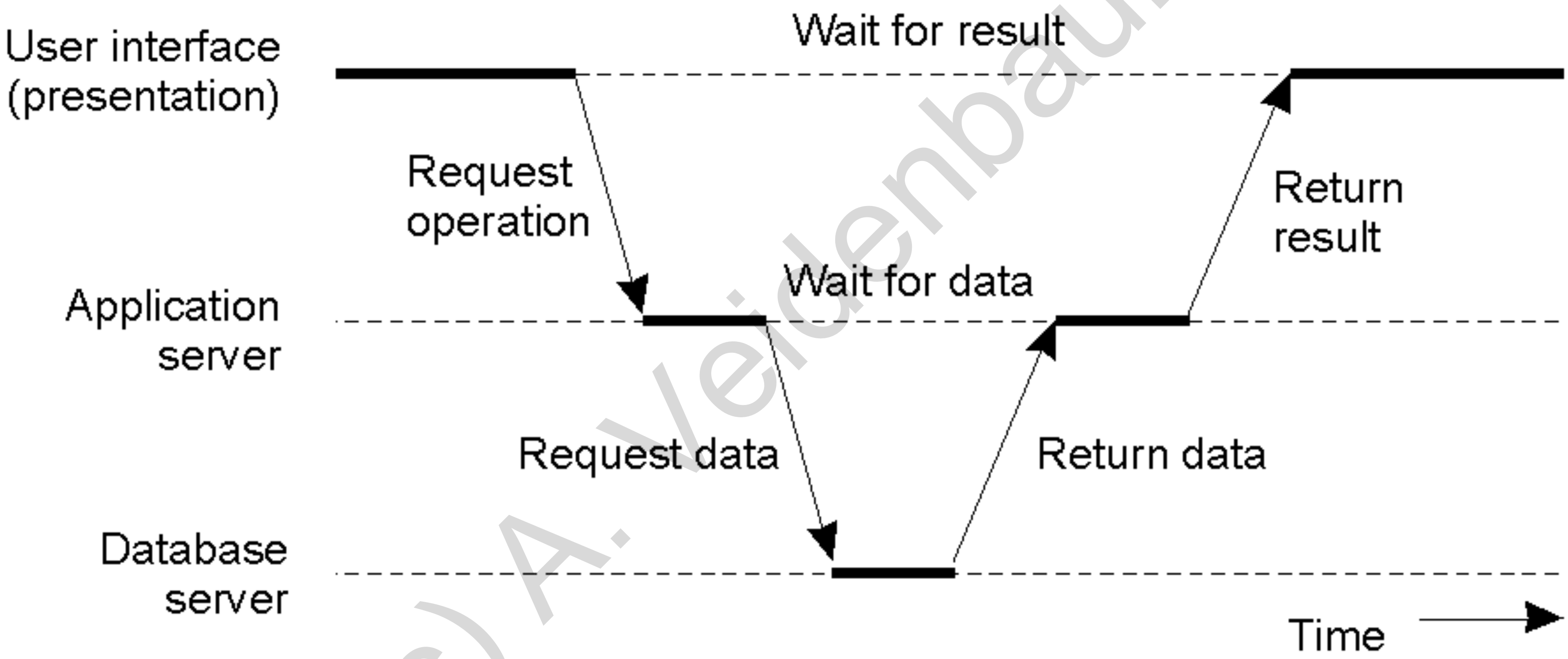
- **Alternative client-server organizations (a) – (e).**



- **Organizations d, e are very popular**
  - **A data base is a backend, just serves requests**
- **Overall, a somewhat limited organization**
  - **A server cannot act as a client!**
- **Solution: three-tiered architecture**
  - **Processing level distributed among servers**
    - » **Common in transaction processing**
      - **Web sites**

Three-tiered Architecture

- **An example of a server acting as a client.**



Decentralized Organizations

- So far we have seen a vertical distribution in a DS
  - Tiers or levels
- Distributed Systems also need replication
  - This leads to a horizontal distribution
    - » Client or server can be physically split into logic parts
      - Each part equivalent but operates on its own share of data
    - » Peer to peer systems are an example
  - The load is distributed, statically or dynamically
- Let us look at peer to peer systems (P2)
  - Processes are all equals in P2P systems
  - But the system function is carried out by all processes
    - » Each can serve as a server and a client (servant)

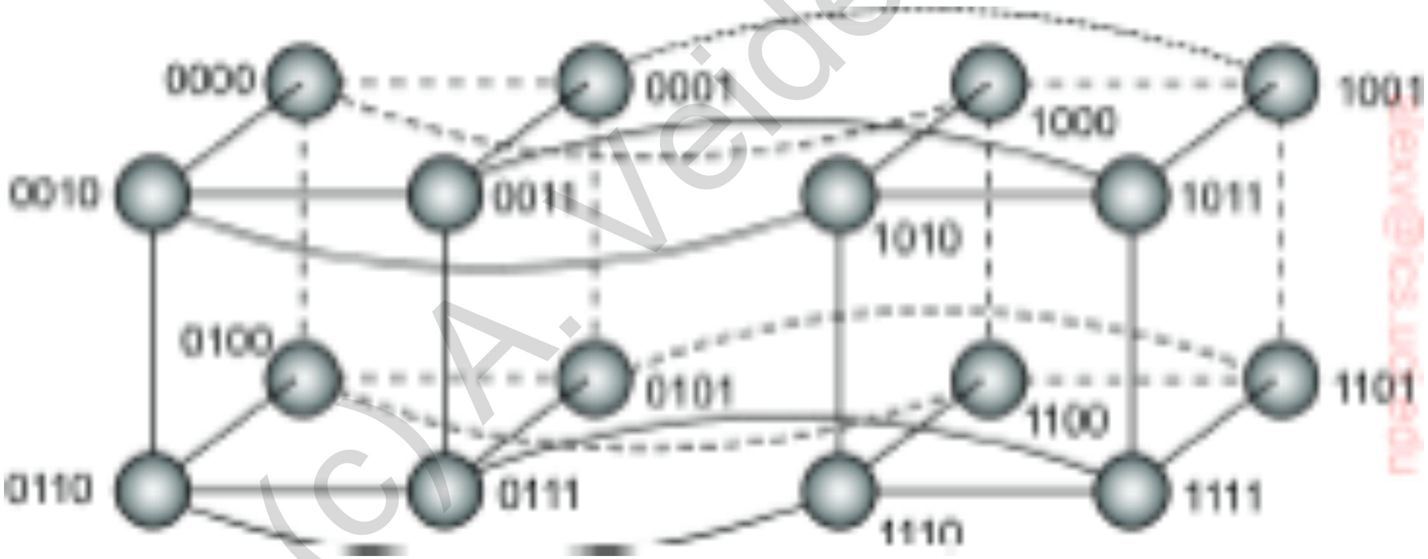
- **The P2P system are symmetric**
- **All processes have to be able to communicate**
  - **All to all type of communication**
- **So it's a question of a communication network**
  - **TCP/IP works just fine**
    - » **Allows arbitrary communication patterns**
    - » **But perhaps slow as there is only one link per system**
- **Sometimes may want to restrict the *all to all***
  - **Overlay networks, where nodes have a limited number of connections to the network**

Structured P2P systems

- **Use a specific, deterministic topology**
  - **Allows efficient data lookup**
- **Generally based on a *semantic-free index***
  - **Each item has a unique key used as an index**
- **One common approach is to use a hash**
  - **key(*data item*) = hash(*data item's value*)**
  - **The whole P2P system now stores (key, value) pairs**
- **Each node is assigned an identifier from the hash space and stores a subset of key with that hash**
- **Access via a lookup function**
  - **existing node = lookup(*key*)**

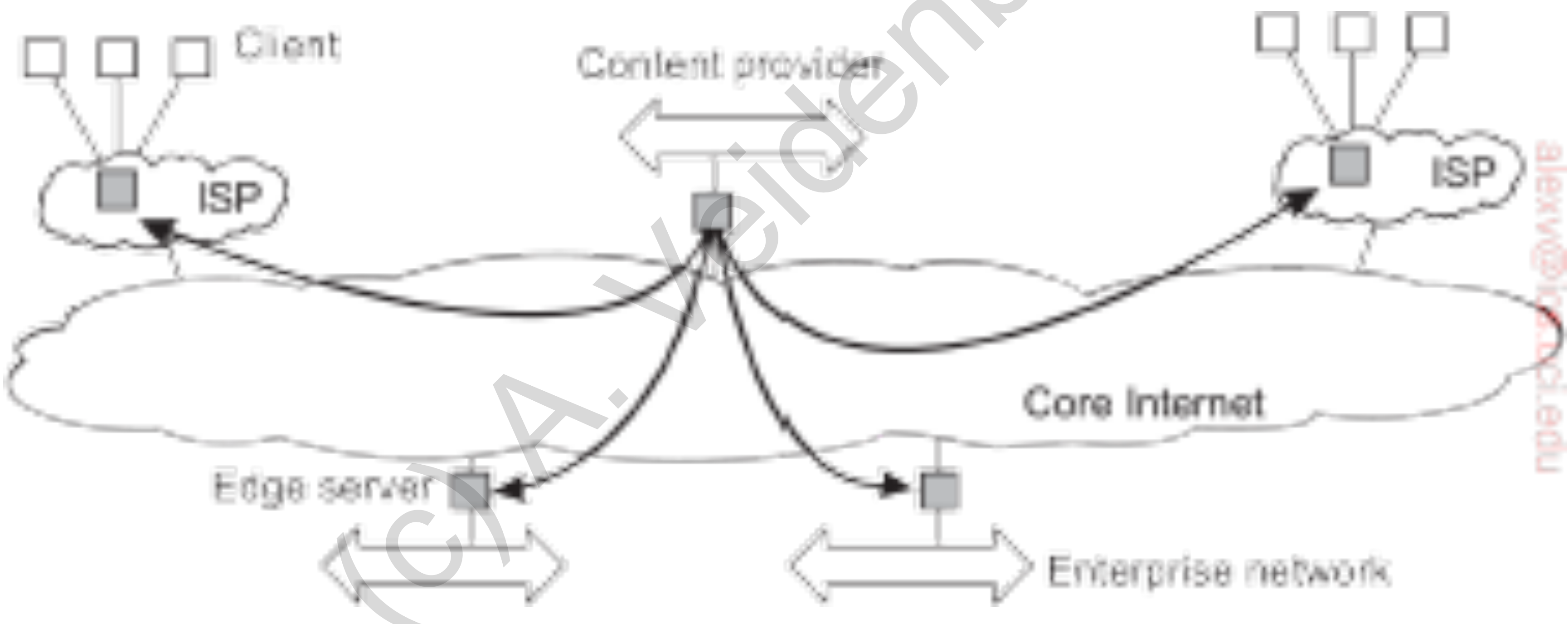
P2P example

- Uses a 4-D hypercube w/ 16 nodes
- A value is hashed to one of the 16 keys
  - A lookup is local
- A node that does not have an item sends the request
  - Toward the node with the correct key



Hybrid Architectures (HA)

- **HAs combine features of client-server and decentralized architectures**
- **An important example are Edge-server systems**
  - **This is how end users connect to the Internet**



Users connect to the internet via edge servers