Client/Server paradigm

ARCOS Group

Distributed Systems

Bachelor In Informatics Engineering

Universidad Carlos III de Madrid

A simple definition

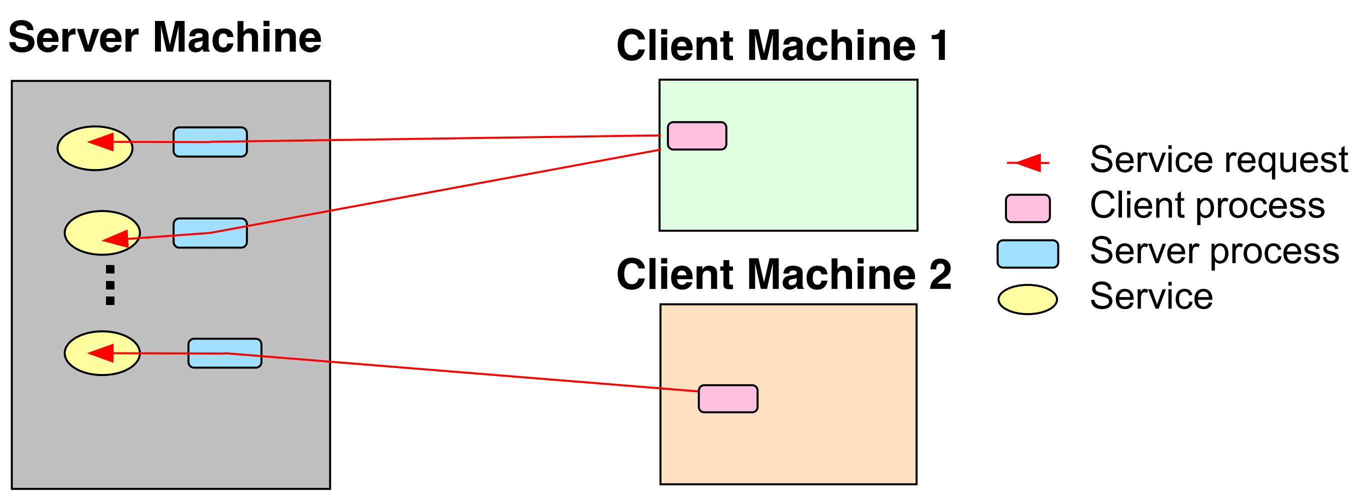
" Server software accepts requests for data from client software and returns the results to the client"

Where are operations executed

- Most of the application processing is done on a computer (client side)
- …which obtains application services (such as database services) from another computer (server side) in a master slave configuration

Client/Server paradigm

- Asigns different roles to the communicating processes
- Server:
  - Offers services
  - Passive: waits for incoming requests from clients
- Client:
  - Requests services
  - Active: sends requests to server(s)



Servers may be…

- Depending on the type of connection with the client:
  - Connection-oriented
  - Connectionless
- Depending on the number of serviced client sessions:
  - Sequential: if it communicates with a single client session at the time
  - Concurrent: if it may communicate with multiple client sessions at the time
- Depending on whether it stores communication state:
  - Stateful
  - Stateless

Connection-oriented servers

- Client and server must establish a connection (logical or physical) before communicating; when finalizing the communication they must close the connection
- Once the connection established it isn't necessary to refer directly to the sender and receiver
- C-O protocols may be seen as stateful – keep track of conversation
- May not serve new client before current one closes the connection!
- E.g.: TCP

Connectionless servers

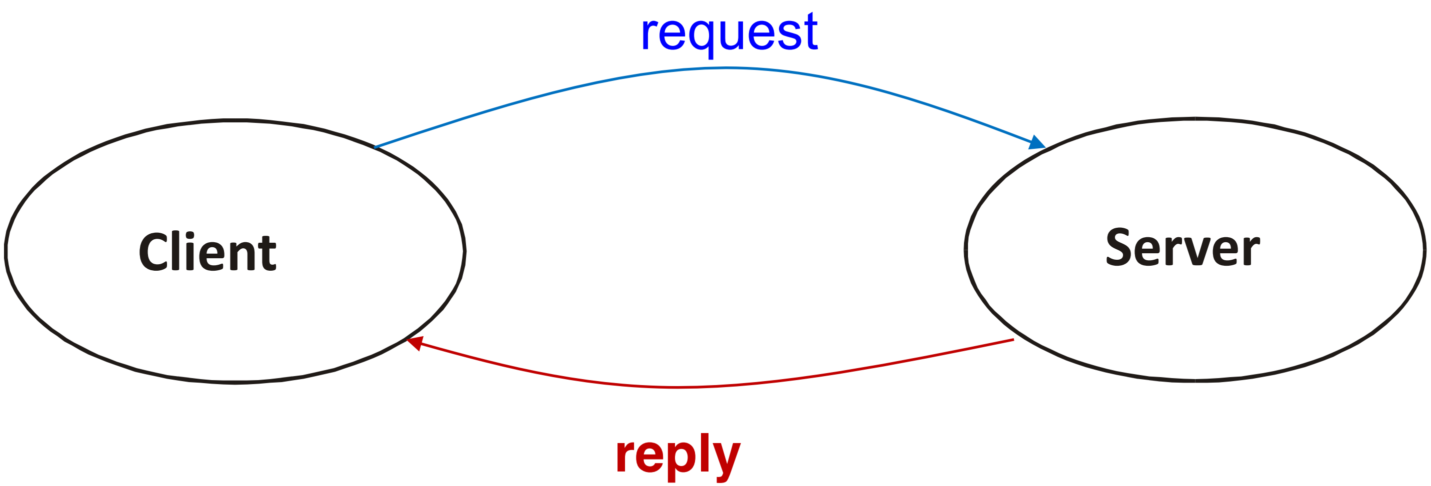
- Data exchanged via self-contained packets which must contain explicit server/client address information
  - No previous agreement
- C-less protocols may be seen as stateless
- May interleave different client requests!
- E.g.: IP, UDP

What is a session?

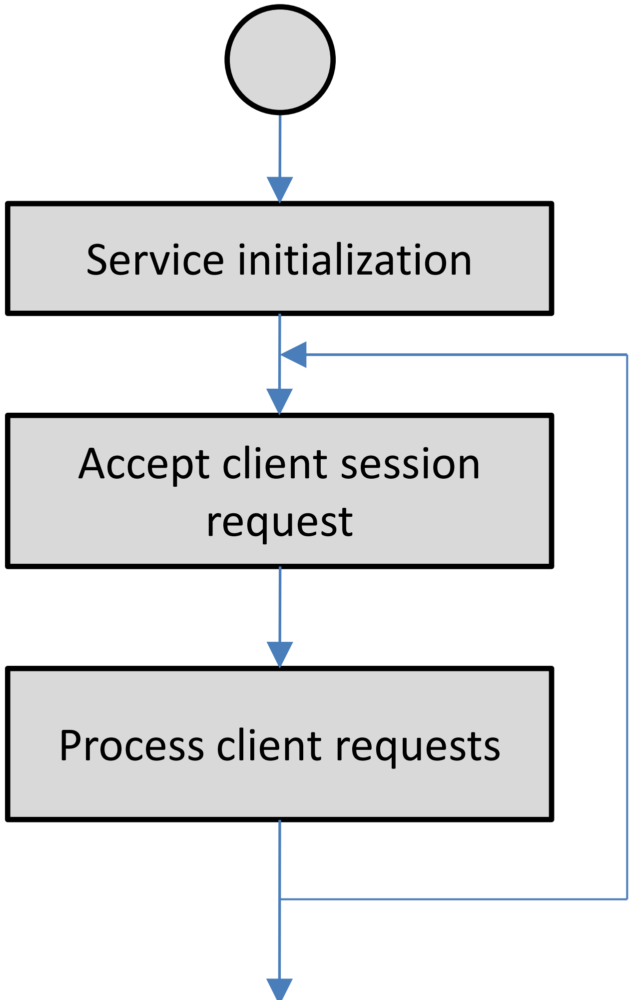
- Session: Interaction between client and server until client gets the requested service
- The server runs an infinit loops which accepts service requests from client sessions
- A service protocol specifies the rules that the client and server follow during a session wrt:
  - Naming a service: services identify themselves via a registered logical name or the server physical process address (machine name + port number)
  - Communication sequence
  - Data representation

Sequential servers

- Serves client requests sequentially
- Does not interleave requests from multiple client sessions
- While attending to a client seesion all others must wait

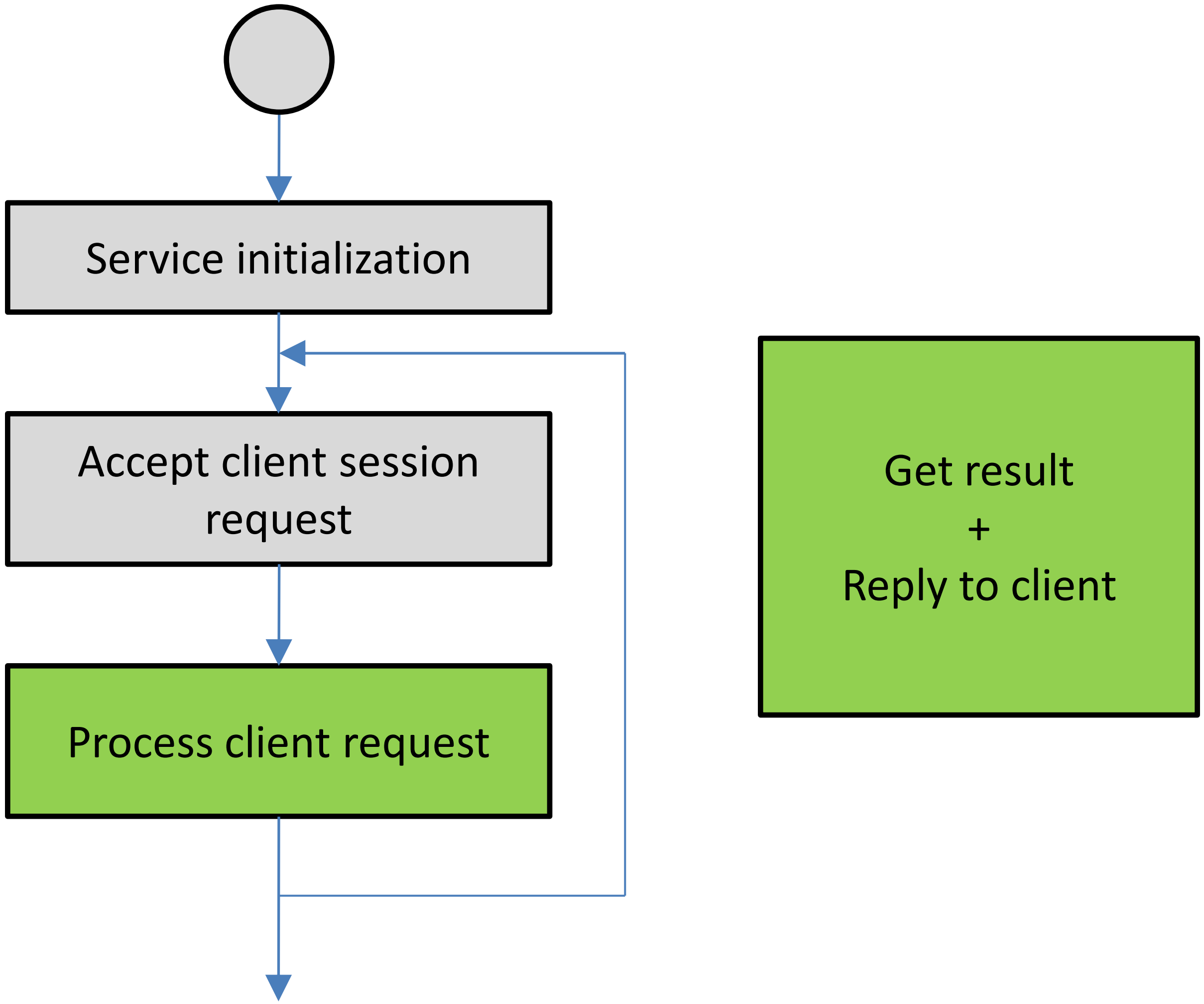


Execution flow



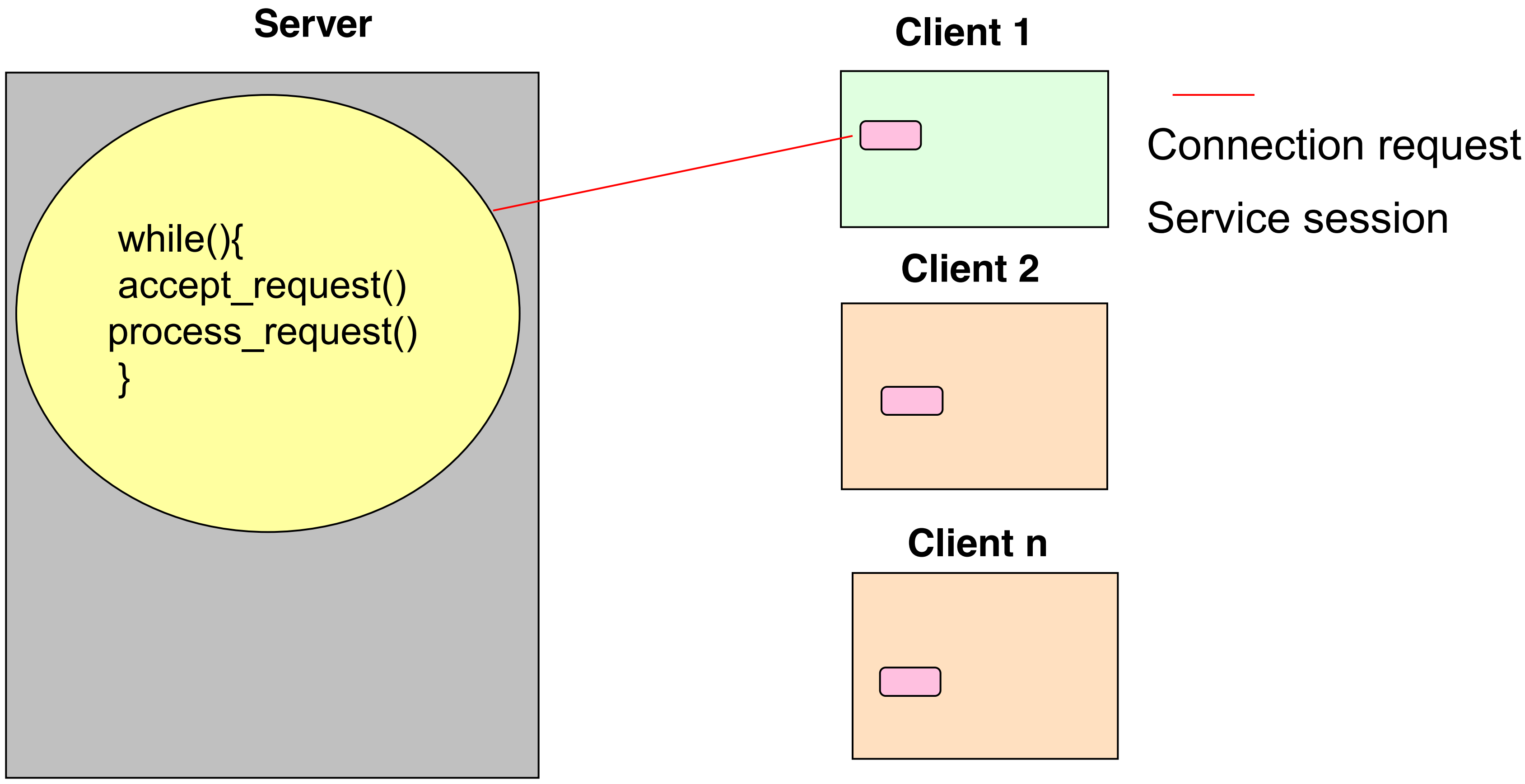
If for each client i keep a service session this server is sequential!

Execution flow

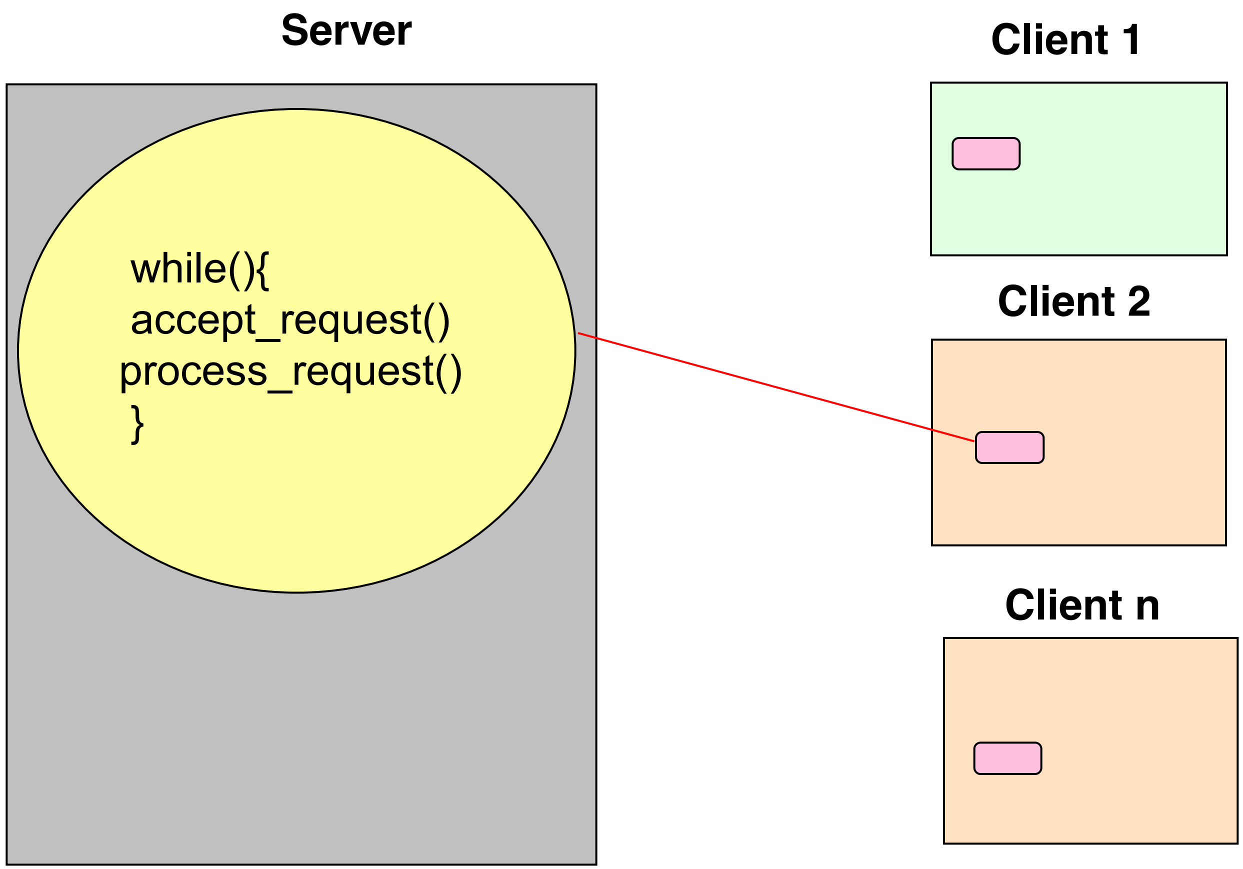


Sequential server

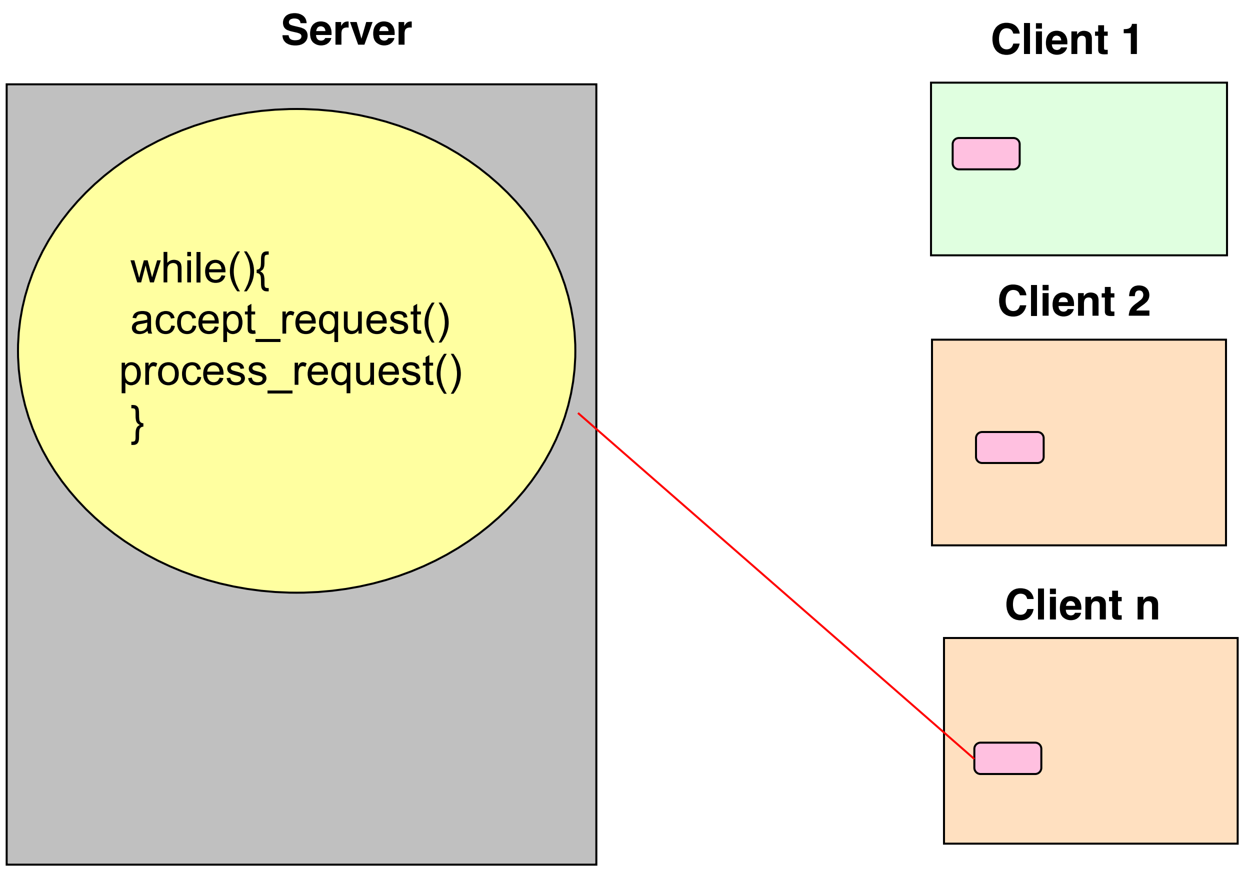
Sequential Client/Server



Sequential Client/Server

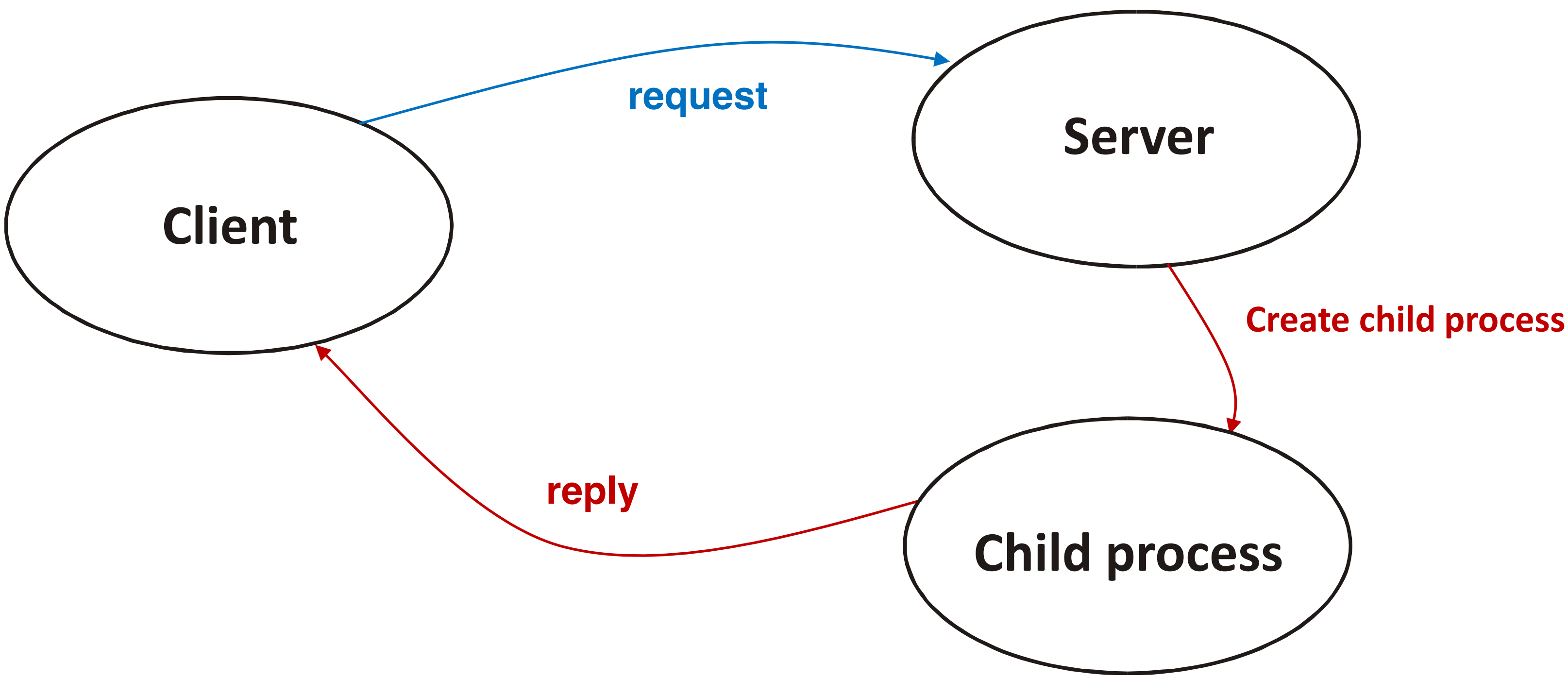


Sequential Client/Server



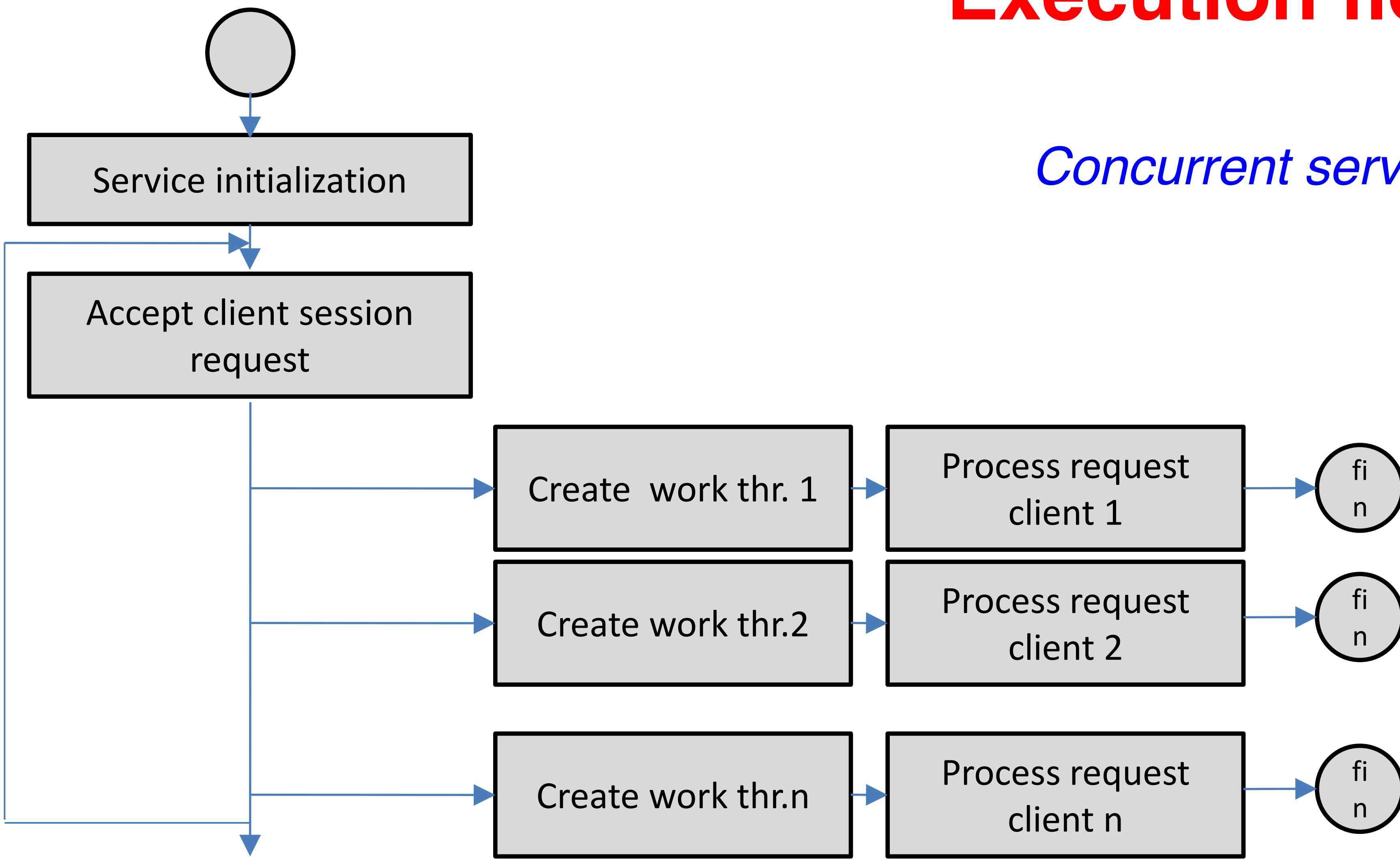
Concurrent servers

- Server creates a child process which will process the request and send the reply to the client
- Multiple client sessions may be interleaved



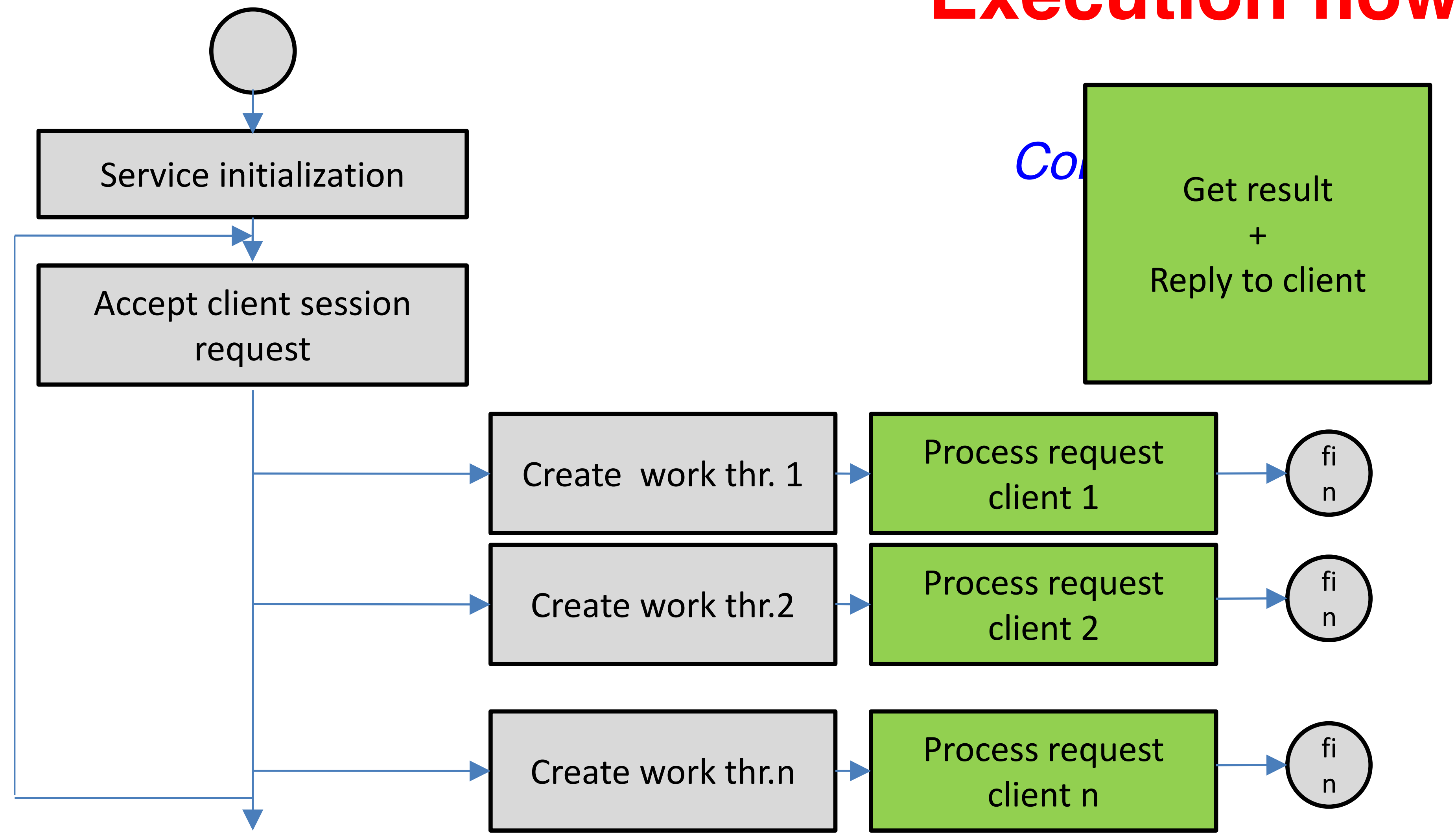
Execution flow

Concurrent server



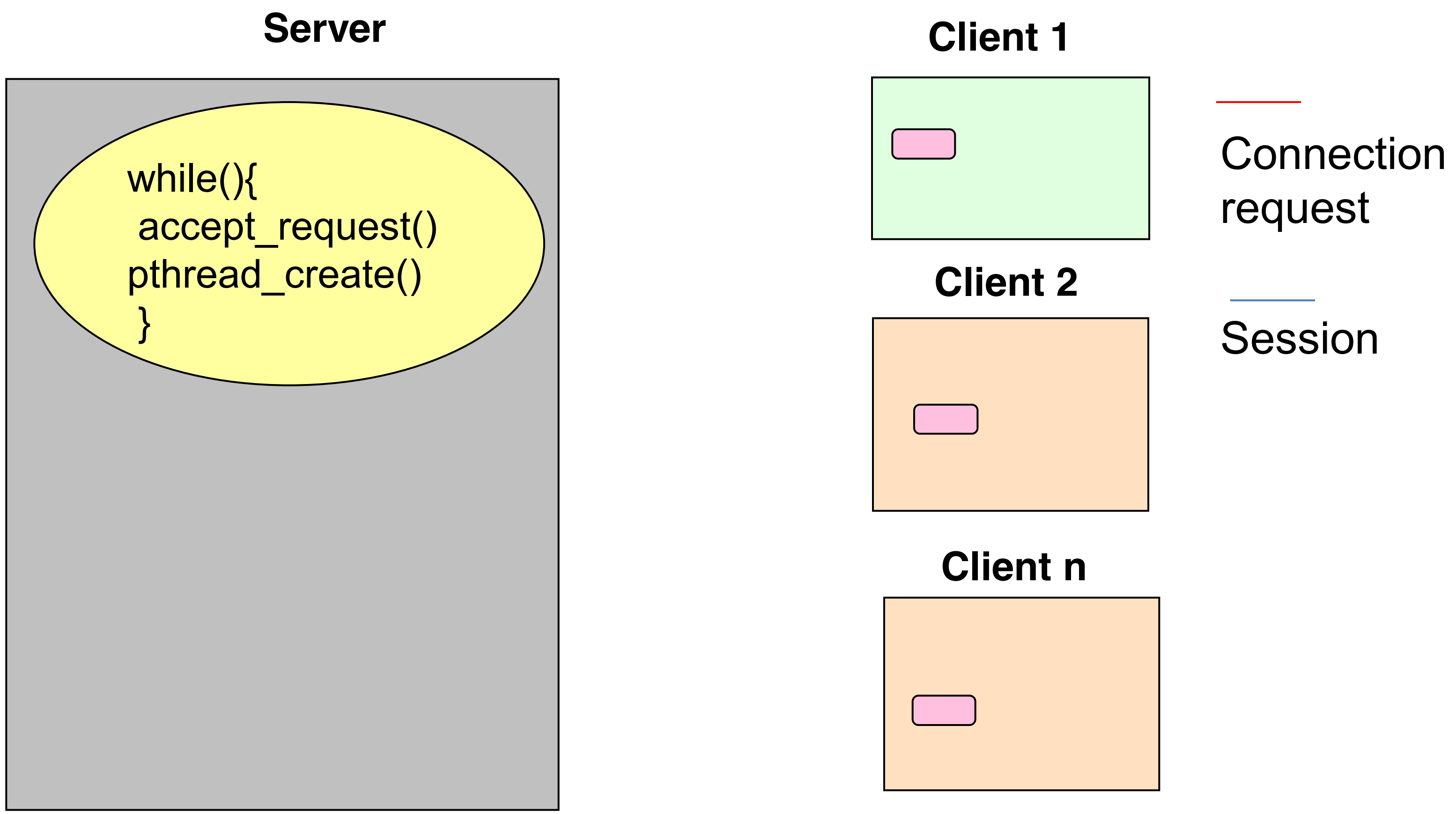
May use asynchronous inter process communication primitives instead of threads!

Execution flow

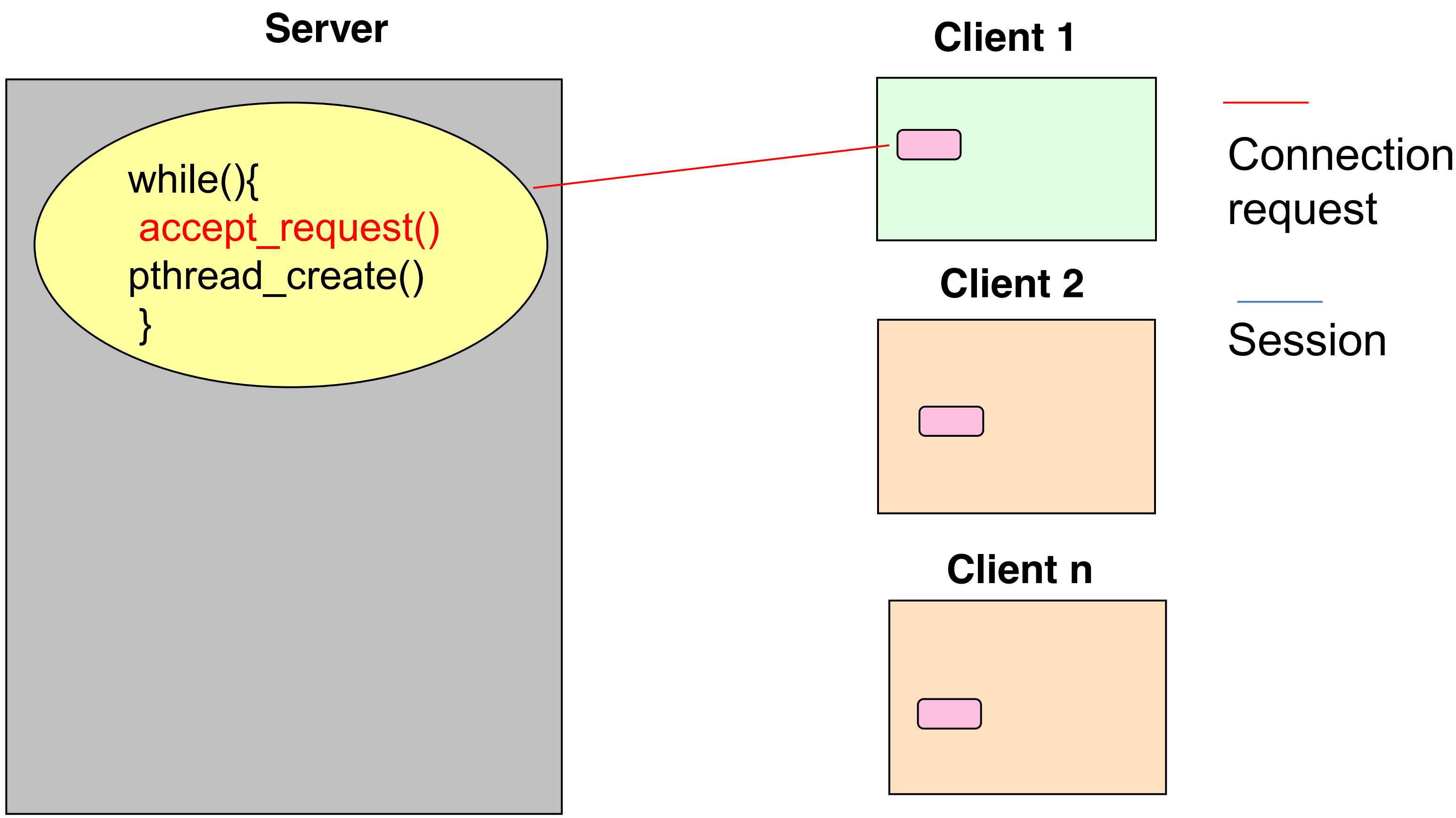


May use asynchronous inter process communication primitives instead of threads!

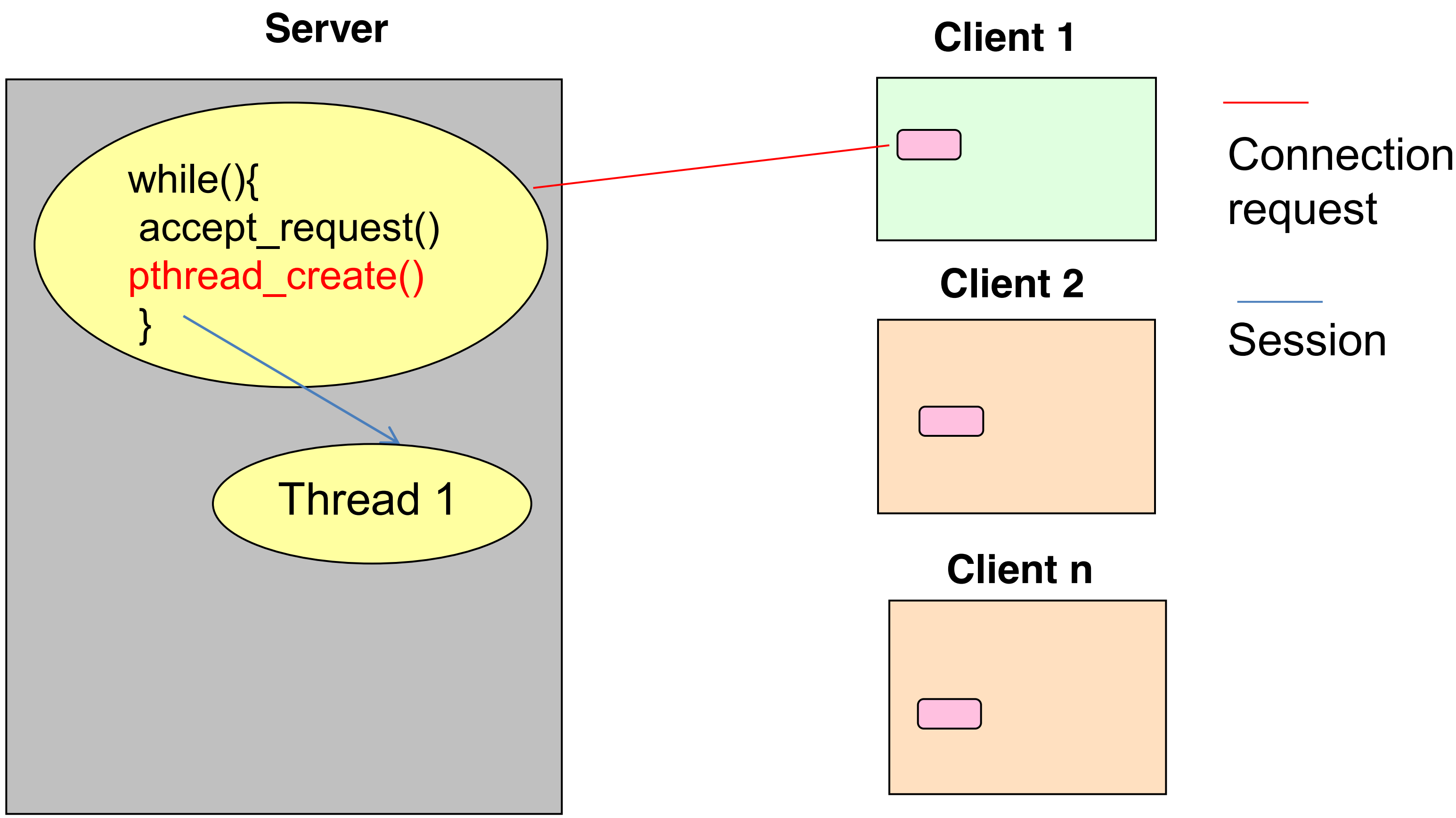
Concurrent Client/Server



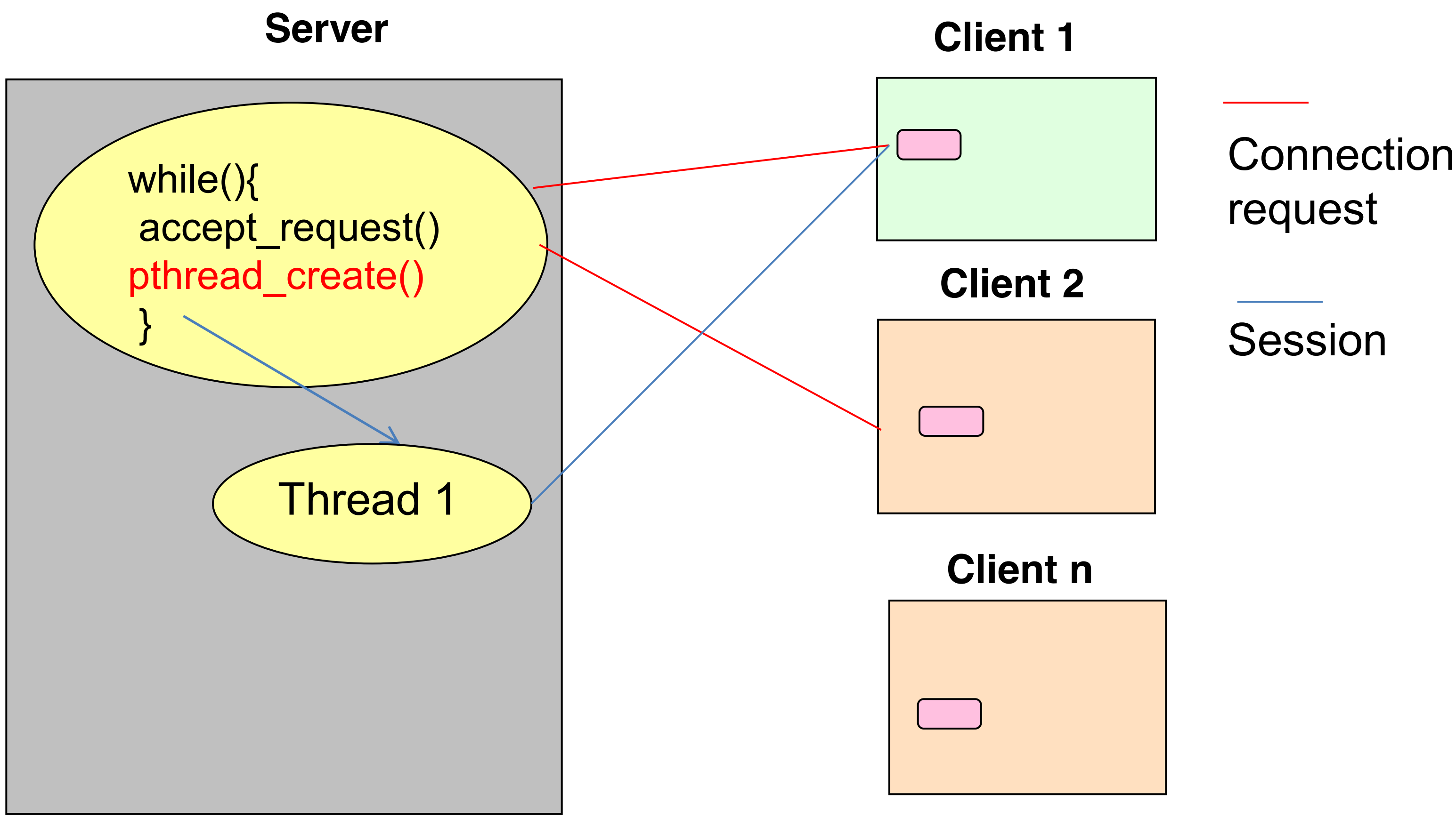
Concurrent Client/Server



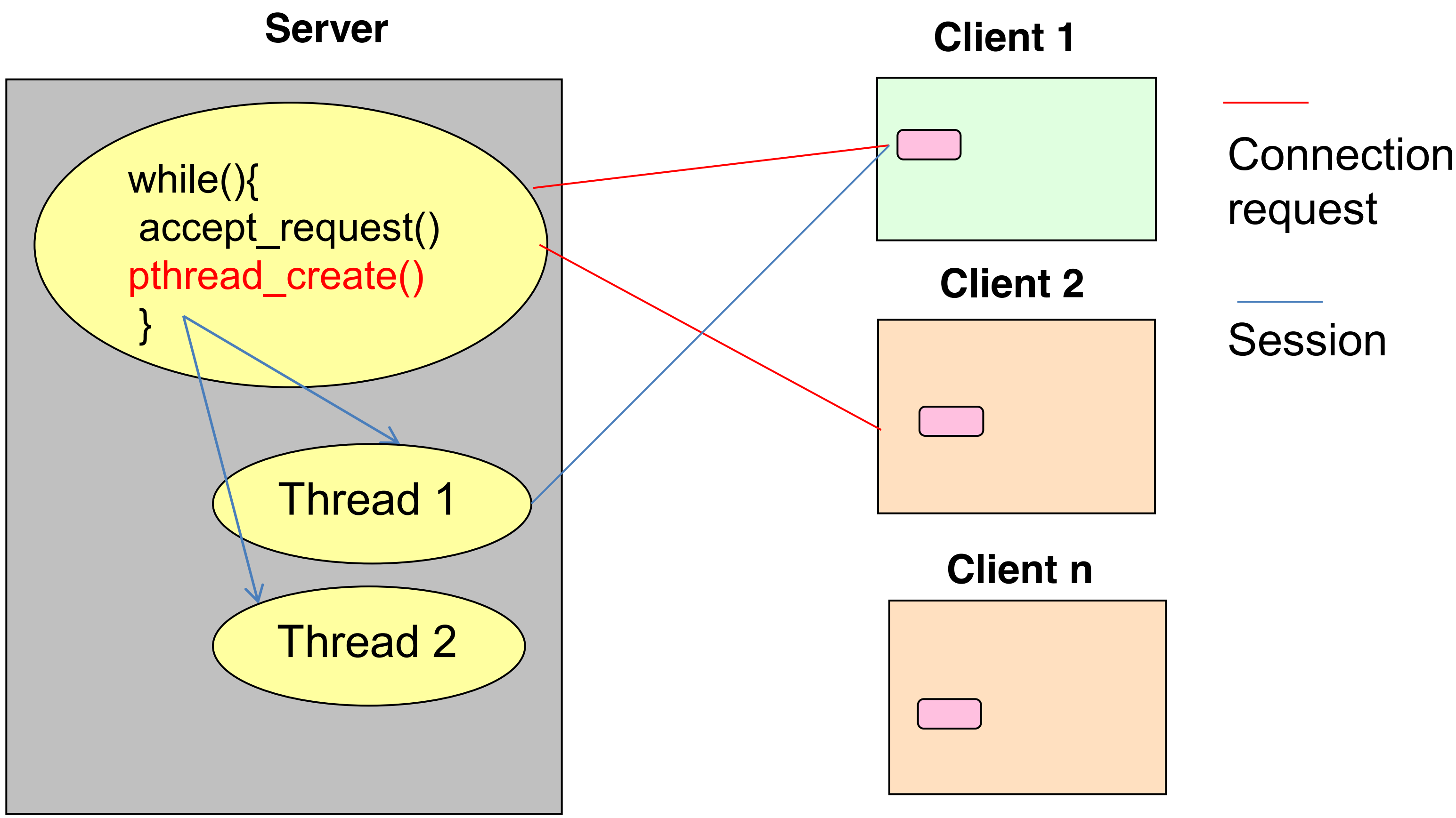
Concurrent Client/Server



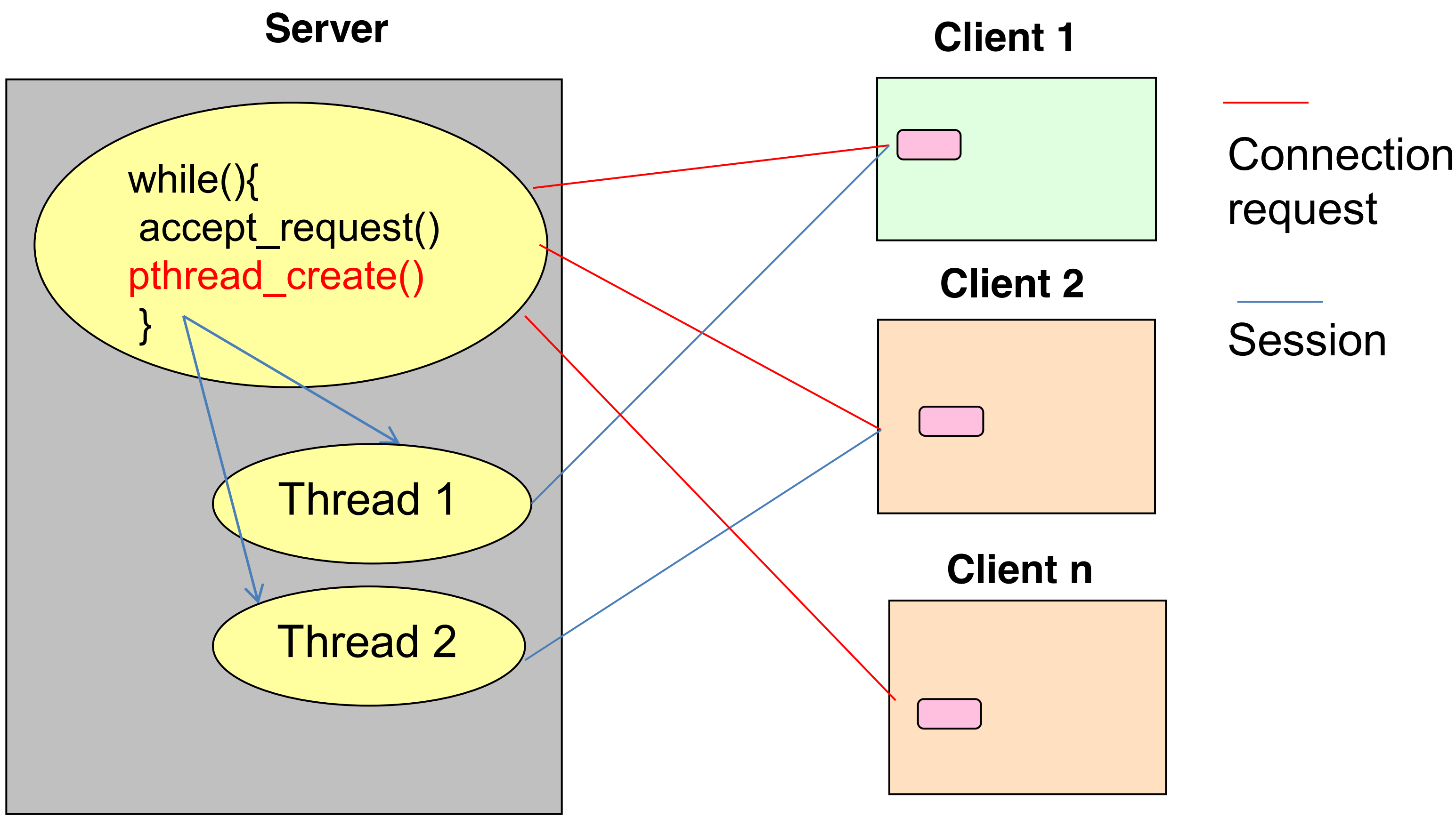
Concurrent Client/Server



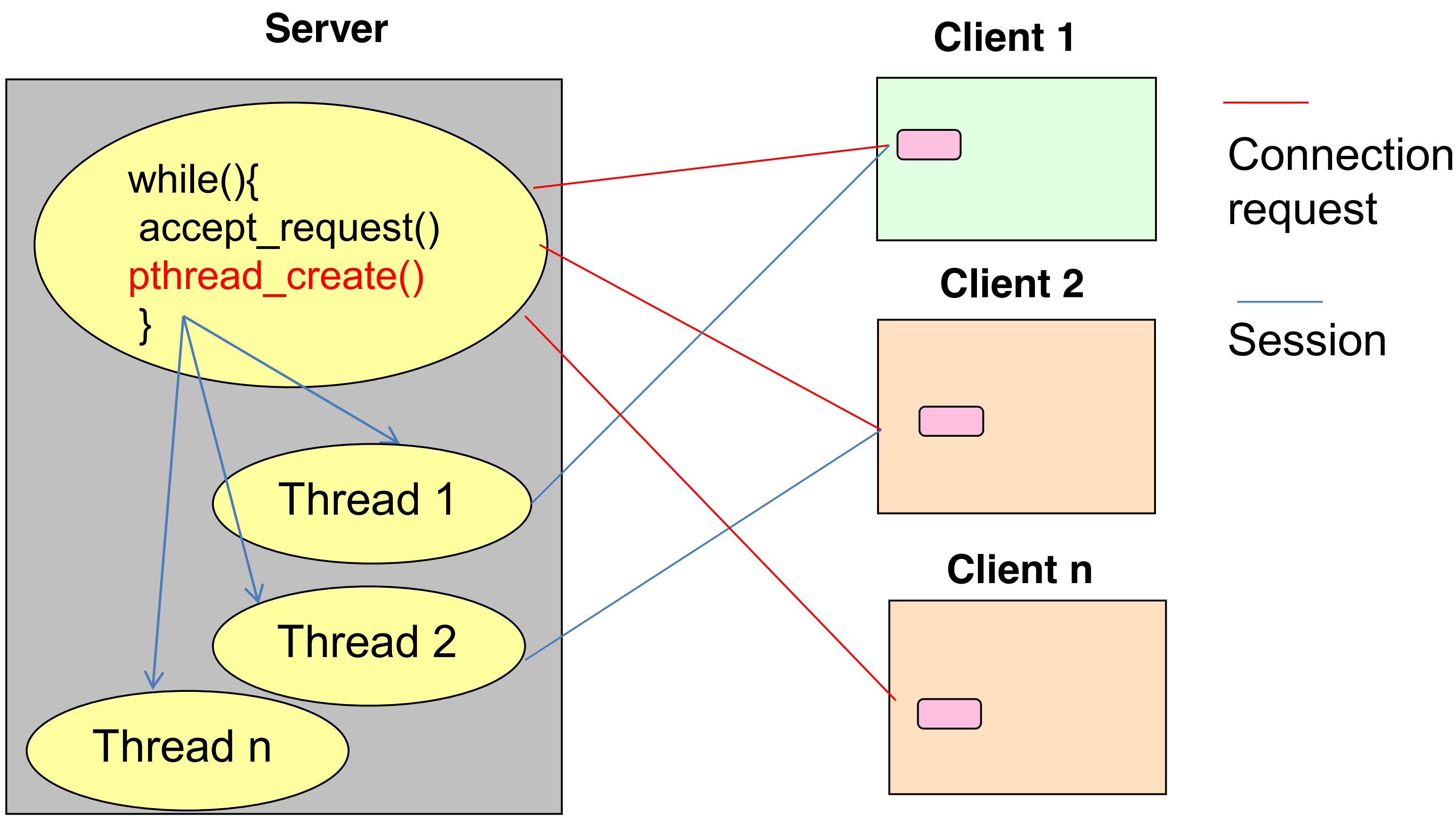
Concurrent Client/Server



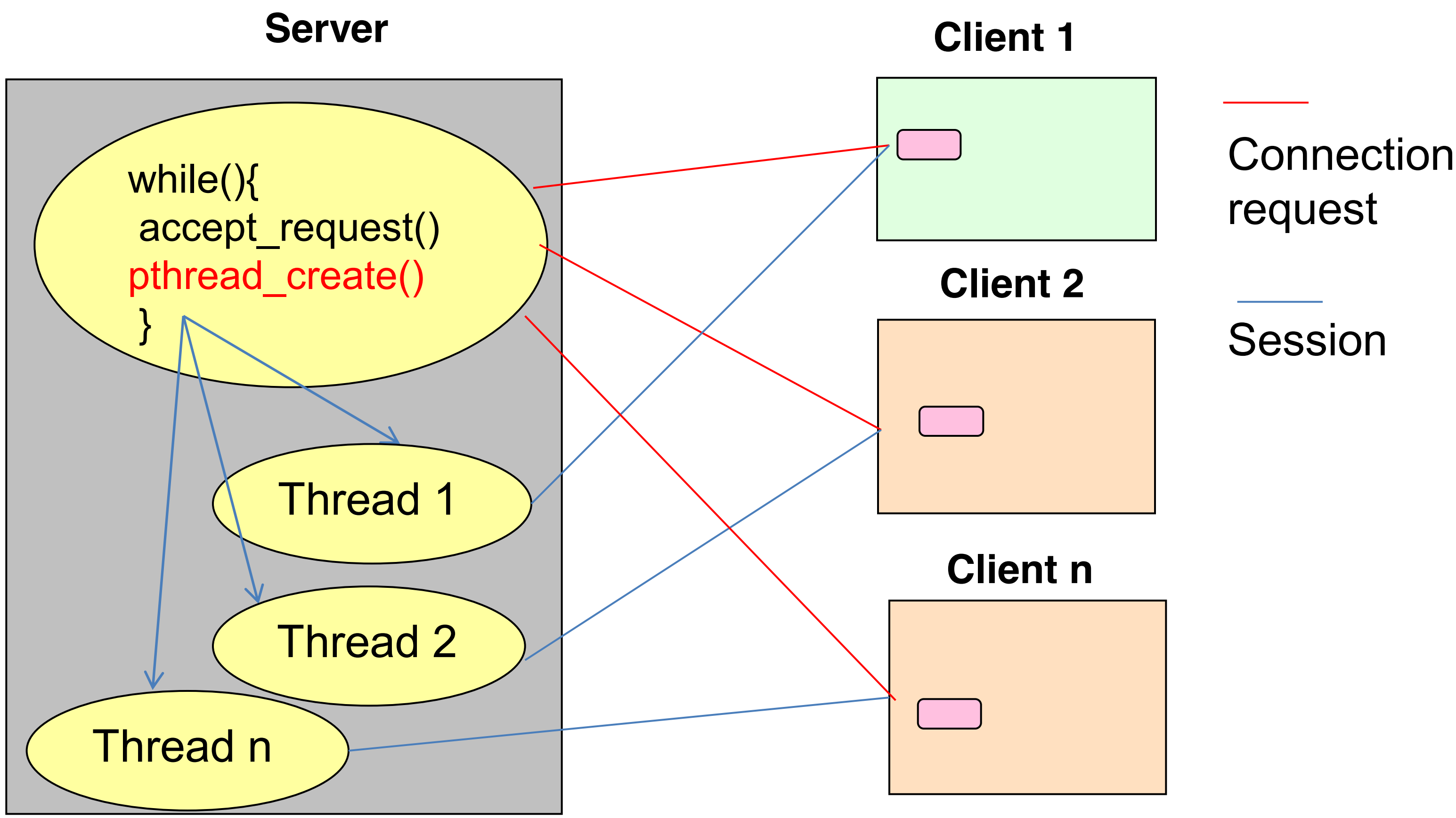
Concurrent Client/Server



Concurrent Client/Server

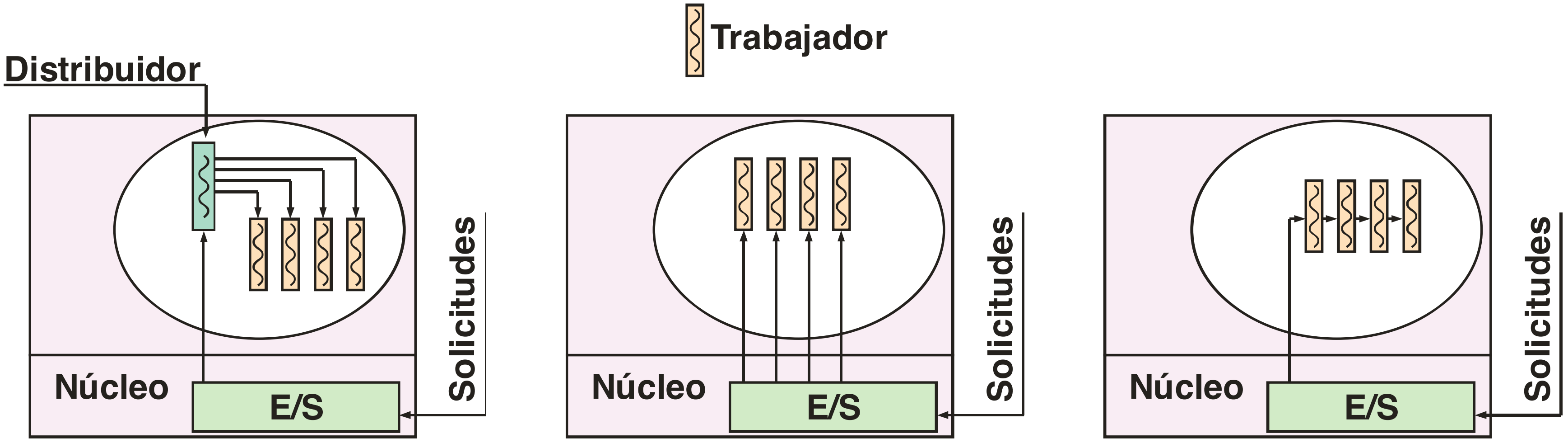


Concurrent Client/Server



Server design using *threads*

- Several alternatives to build parallel servers:
  - A single process which accepts requests and either (1) distributes them to threads from a thread pool or (2) creates a new thread to service the request
  - Set of similar threads which can read requests from a port
  - Pipeline the work and have a specialized thread for each stage



Servers may be…

- Stateless:
  - Every request and reply is independent
  - No state maintained by the server
  - Client may maintain session state and send it as part of the service request to the server
    - Client: “Send me block 1 of file “xxx” from directory “dir”
    - Server: “Here it is”
    - <more of the same>
  - E.g.: HTTP
- Stateful:
  - Maintains state information
  - Each request/reply may depend on previous ones
    - Client: ”Send me file “xxx” from directory “dir”
    - Server: “Here is block 0 of file “xxx”
    - Client: “I have it”
    - Server: “Here is block 1 of file “xxx”
  - E.g.: Telnet

Stateful servers

- Global state:
  - Information common to all clients
  - E.g.: “time of day” server
- Session information
  - Information specific to each client session
  - E.g.: FTP (File Transfer Protocol)

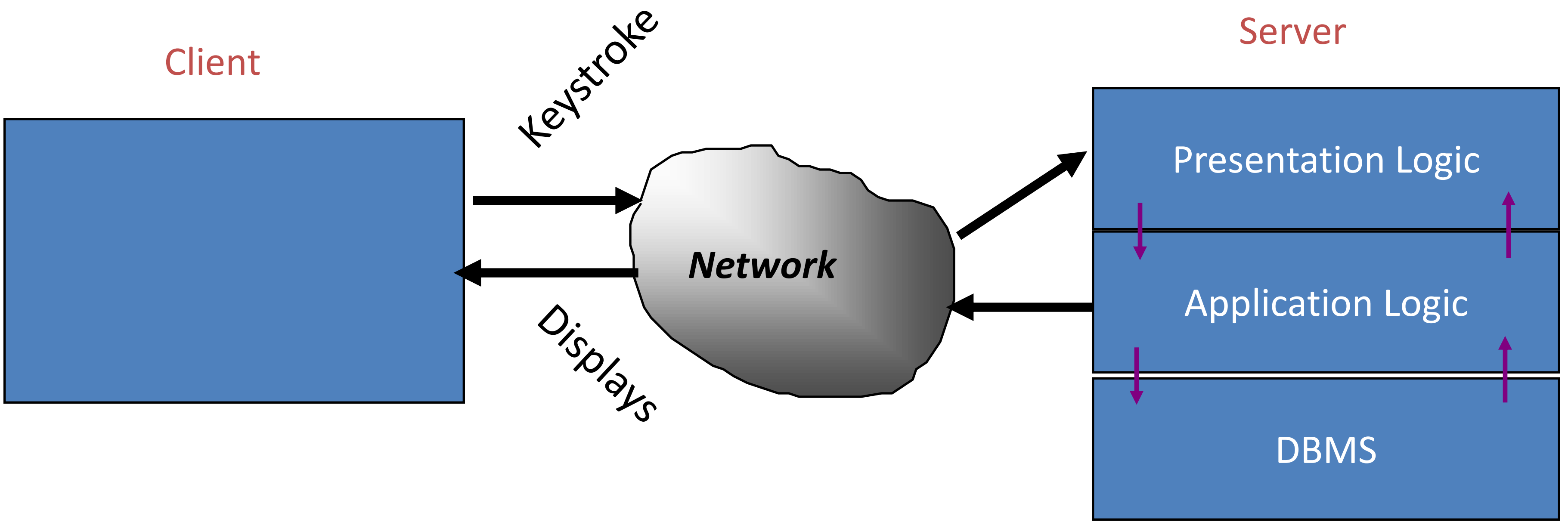
Clients may be…

- Thin: also called lean client or slim client
  - Depends heavily on other computer (e.g. its server) to process most or all of its business logic - which traditional systems like fat clients take on
    - E.g. : the server may need to provide data persistence, process information on client's behalf, etc
  - May be seen as amortizing computing services across several user interfaces
  - Problem: server become single point of failure!
    - Good for checking security thread models
    - Bad if denial of service attack from a client
- Thick: also called fat client or heavy client

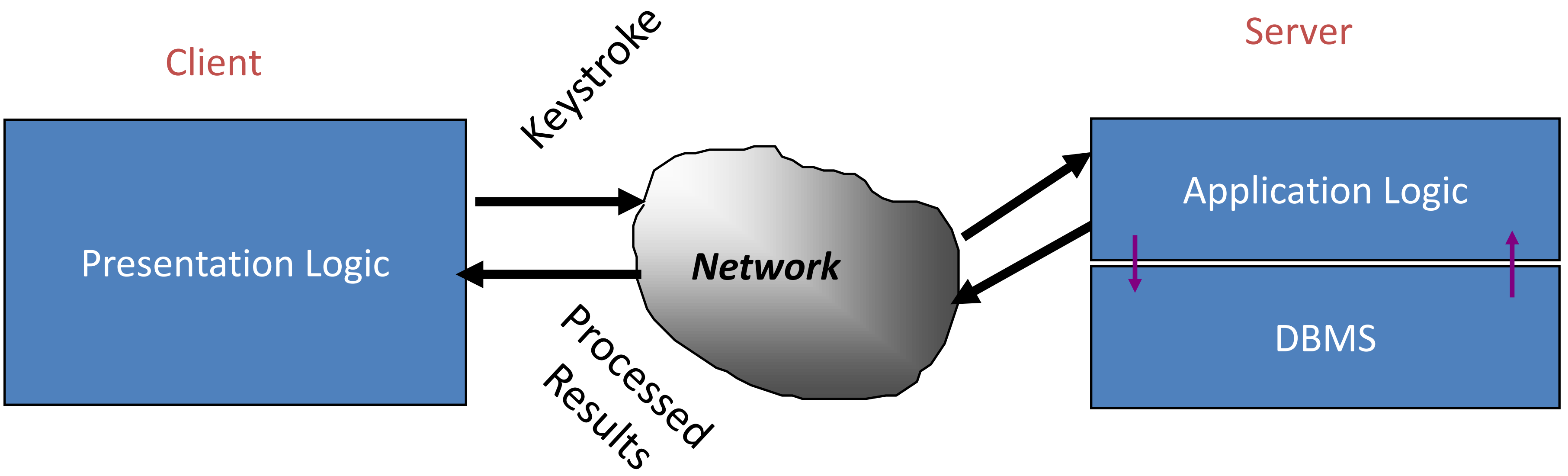
SW architecture

- Usually consists of three layers/tiers:
  - Presentation: user interface issues
  - Application logic: isolates data processing in one location and maximizes reuse, modification in services does not affect presentation
    - Server needs to process client request, compute result and return it to client
    - Client needs to send service request and visualise result
  - Services - we need two types:
    - On server - those processing the request
    - Some IPC mechanism!
    - Must be able to manage data
- May seem similar but different from MVC architecture!
  - View sends updates to controller; controller updates the model, view gets updated directly from model
  - Model = data +domain logic (+persistence, notification)
  - View = query model, render view
  - Controller = init model, wiring up events between controller and V/M

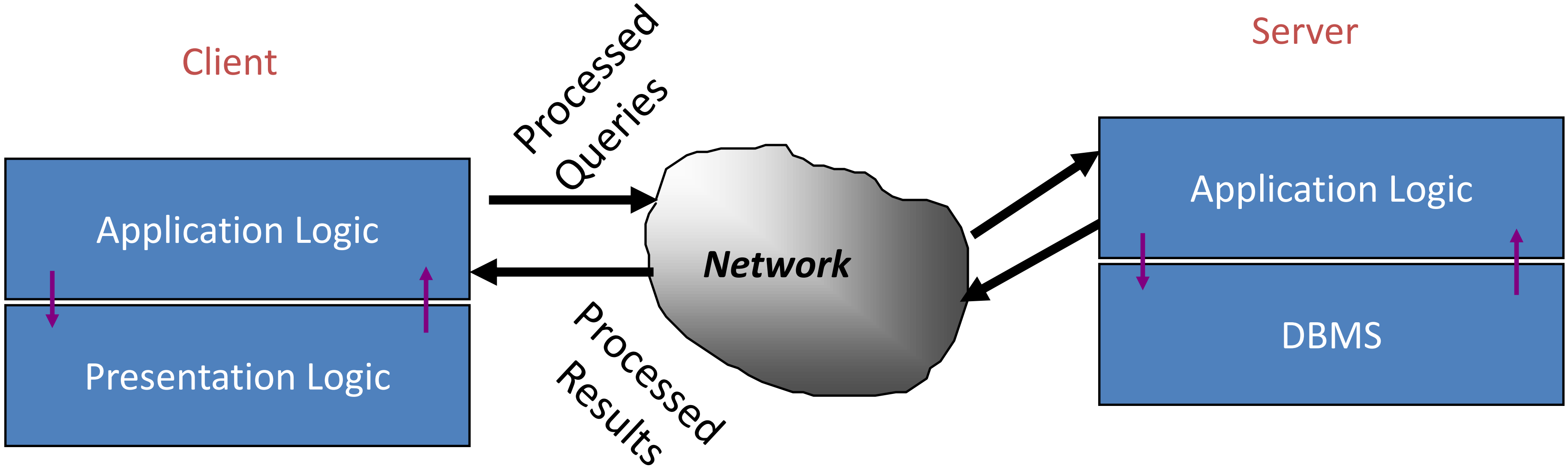
Client (dumb) - Server Model



True Client-Server Model



Distributed Client-Server Model

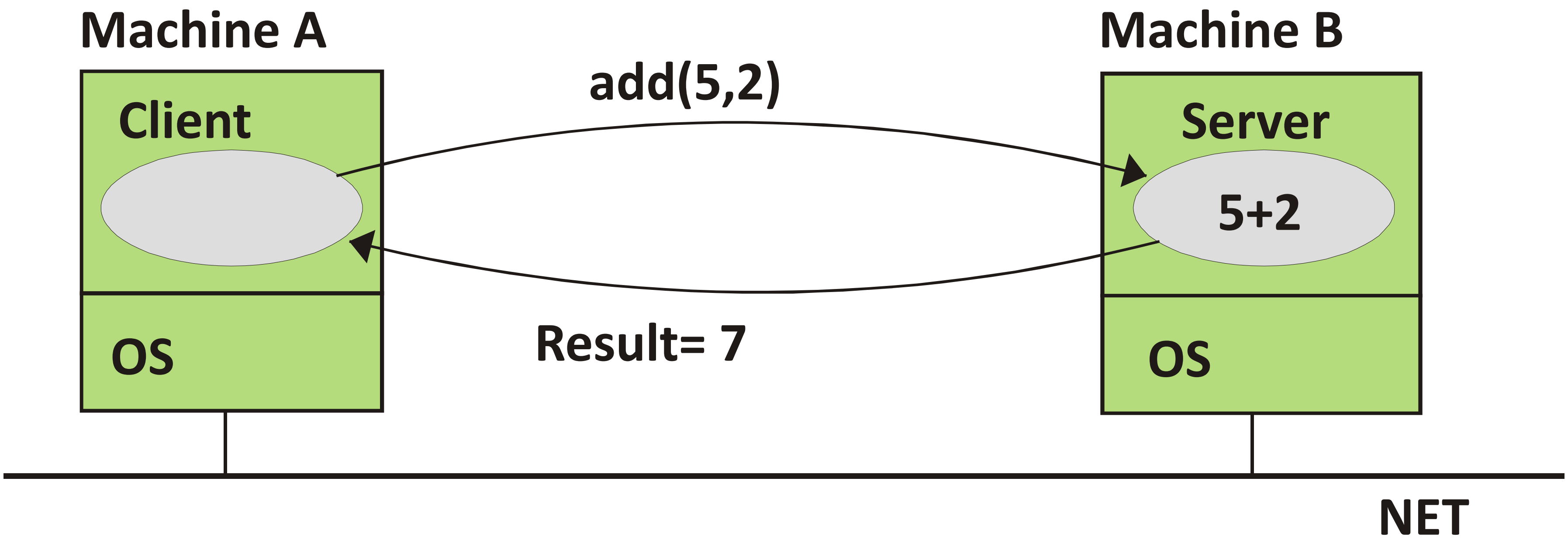


Typical to fat clients
If too complex may want to split into a three-tier

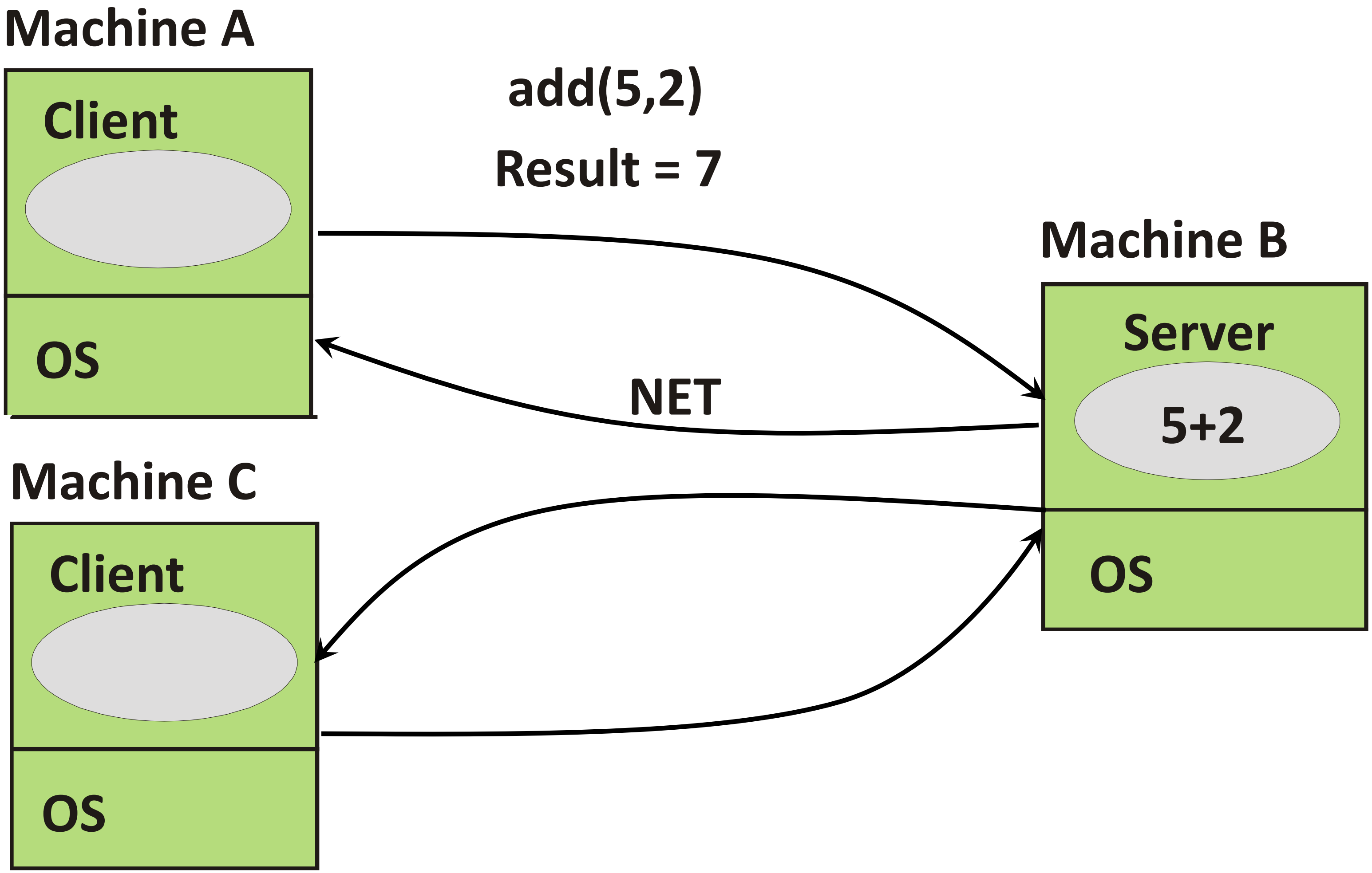
Client/Server applications using message queues

- Distributer thread:
  - Each request results in creating a new work thread which:
    - Processes the request
    - Sends reply to client
  - When client session finishes the thread is destroyed
- Concurrent model:
  - Distributer and work threads execute concurrently

Example: add two numbers



Example: add two numbers

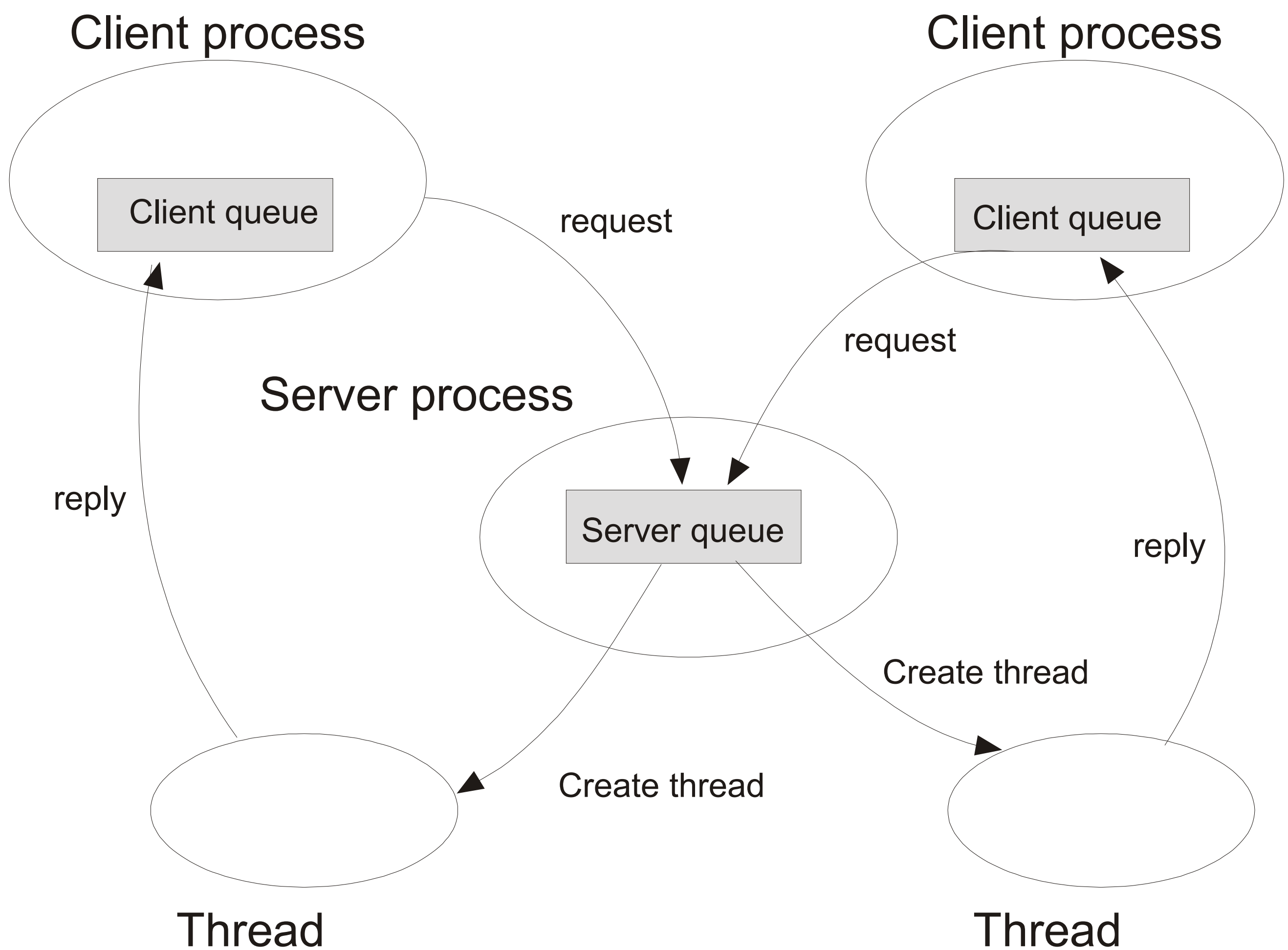


Example: data types

```
#define MAXSIZE        256

struct request  {
   int a;                      /* op. 1 */
   int b;                      /* op. 2 */
   char q_name[MAXSIZE];       /* client queue name - this is
                               where the server sends the reply
                               to */
};
```

The structure of a multithread server



Multithread server with message queues (I)

```
#include "mensaje.h"
#include <mqueue.h>
#include <pthread.h>
#include <stdio.h>

/* mutex and condition variables for the message copy */
pthread_mutex_t mutex_msg;
int msg_not_copied = TRUE;          /* TRUE = 1 */
pthread_cond_t cond_msg;

int main(void)
{
   mqd_t q_server;                  /* server queue */
   struct request msg;              /* message to receive */
   struct mq_attr q_attr;           /* queue atributes */
   pthread_attr_t t_attr;           /* thread atributes */

   q_attr.mq_maxmsg = 20;
   q_attr.mq_msgsize = sizeof(struct request));
```

Multithread server with message queues (II)

```
q_server = mq_open("SERVER", O_CREAT|O_RDONLY, 0700, &attr);
 if (q_server == -1) {
       perror("Can't create server queue");
       return 1;
 }

 pthread_mutex_init(&mutex_msg, NULL);
 pthread_cond_init(&cond_msg, NULL);
 pthread_attr_init(&attr);

 /* thread atributes */
 pthread_attr_setdetachstate(&t_attr, PTHREAD_CREATE_DETACHED);
```

Multithread server with message queues (III)

```
 while (TRUE) {
        mq_receive(q_server, &msg, sizeof(struct request), 0);

        pthread_create(&thid, &attr, process_message, &msg);









   }
}
```

Multithread server with message queues (IV)

```
while (TRUE) {
        mq_receive(q_server, &msg, sizeof(struct request), 0);

        pthread_create(&thid, &attr, process_message, &msg);



    }
}
```

Race condition!!

Multithread server with message queues (V)

```
  while (TRUE) {
        mq_receive(q_server, &msg, sizeof(struct request), 0);

        pthread_create(&thid, &attr, process_message, &msg);

        /* wait for thread to copy message */
        pthread_mutex_lock(&mutex_msg);
        while (message_not_copied)
               pthread_cond_wait(&cond_msg, &mutex_msg);
        message_not_copied = TRUE;
        pthread_mutex_unlock(&mutex_msg);

   }  /* FIN while */
} /* Fin main */
```

Critical section

Multithread server with message queues (VI)

```
void process_message(struct mensaje *msg){
   struct request msg_local;          /* local message */
   struct mqd_t q_client;      /* client queue */
   int result;

   /* thread copies message to local message*/
   pthread_mutex_lock(&mutex_msg);
   memcpy((char *) &msg_local, (char *)&msg, sizeof(struct
        request));

   /* wake up server */
   message_not_copied = FALSE;        /* FALSE = 0 */

   pthread_cond_signal(&cond_msg);

   pthread_mutex_unlock(&mutex_msg);
```

Multithread server with message queues (VII)

```
/* execute client request and prepare reply */
result = msg_local.a + msg_local.b;

/* return result to client by sending it to queue  */
q_client = mq_open(msg_local.name, O_WRONLY);

if (q_client == -1)
     perror("Can't open client queue */
else {
     mq_send(q_client, (char *) &result, sizeof(int), 0);
     mq_close(q_client);
}
pthread_exit(0);
}
```

Client process

```
#include "mensaje.h"
#include <mqueue.h>
void main(void) {
   mqd_t q_server;          /* server message queue */
   mqd_t q_client;          /* client message queue */

   struct request req;
   int res;
   struct mq_attr attr;

   attr.mq_maxmsg = 1;
   attr.mq_msgsize = sizeof(int);
   q_client = mq_open("CLIENT_ONE", O_CREAT|O_RDONLY, 0700, &attr);

   q_server = mq_open("ADD_SERVER", O_WRONLY);

   /* fill in request */
   req.a = 5;     req.b = 2;   strcpy(req.q_name, "CLIENT_ONE");

   mq_send(q_server, &req, sizeof(struct request), 0);
   mq_receive(q_client, &res, sizeof(int), 0);

   mq_close(q_server);
   mq_close(q_client);
   mq_unlink("CLIENT_ONE");
}
```