Grupo ARCOS
Universidad Carlos III de Madrid

Lección 3
Ejercicios de paso de mensajes

Sistemas Distribuidos
Grado en Ingeniería Informática

Ejercicio 2

Se desea diseñar un modelo de vector distribuido.
Sobre un vector distribuido se definen los siguientes servicios:

- int **init** (char *nombre, int N). Este servicio permite inicializar un array distribuido de N números enteros. La función devuelve 1 cuando el array se ha creado por primera vez. En caso de que el array ya esté creado, la función devuelve 0. La función devuelve -1 en caso de error.
- int **set** (char *nombre, int i, int valor). Este servicio inserta el valor en la posición i del array nombre. Devuelve -1 en caso de error.
- int **get** (char*nombre, int i, int *valor). Este servicio permite recuperar el valor del elemento i del array nombre. Devuelve -1 en caso de error.

Diseñe un sistema distribuido que implemente el servicio con colas POSIX de forma que permita trabajar con varios clientes concurrentemente.

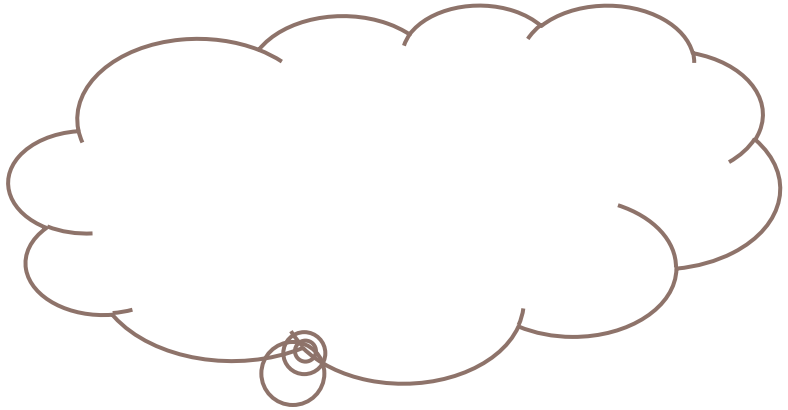
Ejercicio 2

Se desea diseñar un modelo de vector distribuido.
Sobre un vector distribuido se definen los siguientes servicios:

- int **init** (char *nombre, int N). Este servicio permite inicializar un array distribuido de N números enteros. La función devuelve 1 cuando el array se ha creado por primera vez. En caso de que el array ya esté creado, la función devuelve 0. La función devuelve -1 en caso de error.
- int **set** (char *nombre, int i, int valor). Este servicio inserta el valor en la posición i del array nombre. Devuelve -1 en caso de error.
- int **get** (char*nombre, int i, int *valor). Este servicio permite recuperar el valor del elemento i del array nombre. Devuelve -1 en caso de error.

Diseñe un **sistema distribuido** que implemente el servicio con **colas POSIX** de forma que permita trabajar con varios clientes **concurrentemente**.

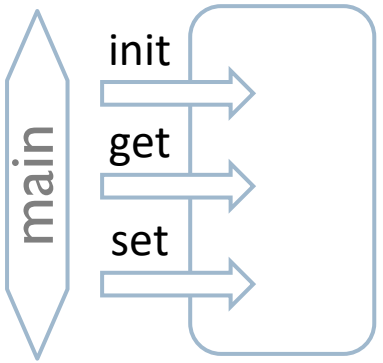
Diseño progresivo



1. Sistema
2. Distribuido
3. Con colas de mensajes POSIX
4. Concurrentes

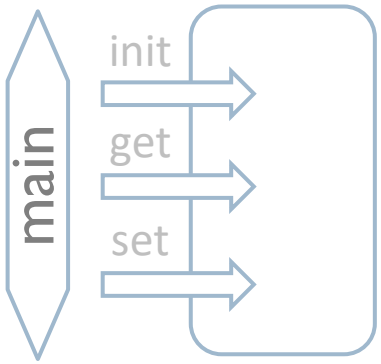
Diseño NO distribuido (v0.2)

Biblioteca usada desde programa



1. **Sistema**
2. Distribuido
3. Con colas de mensajes POSIX
4. Concurrentes

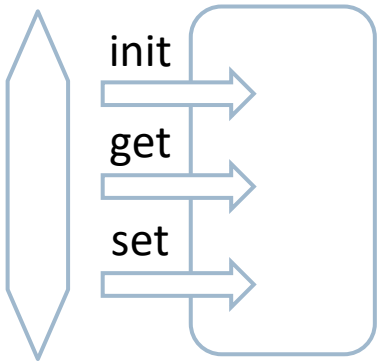
Diseño no distribuido…



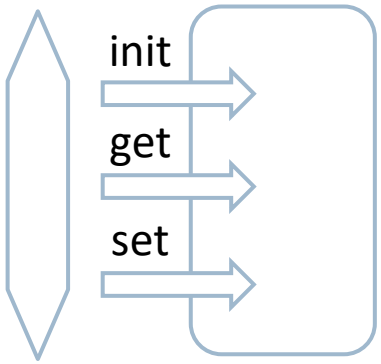

```
#include <stdio.h>
#include <stdlib.h>
#include "lib.h"

int main ( int argc, char *argv[] )
{
    int ret,  val ;

    ret = init ("nombre", 10) ;
    if (ret < 0) {  printf("ERROR: init with code %d\n", ret); exit(-1);  }

    ret = set ("nombre", 1,  0x123) ;
    if (ret < 0) {  printf("ERROR: set with code %d\n", ret);   exit(-1);   }

    ret = get ("nombre", 1, &val) ;
    if (ret < 0)         {  printf("ERROR: get with code %d\n", ret);              exit(-1);   }
    if (val != 0x123) {  printf("ERROR: get %d but set %d\n", 0x123, val);   exit(-1);   }

    printf("OK\n") ;
    return 0 ;
}
```

Diseño no distribuido…

```
int      a_neltos = 0 ;
int   * a_values[100] ; // = [ [0…N1], [0…N2], … [0…NN] ] ;
char * a_keys  [100] ; // = [ “key1”,    “key2”,   …   “keyN” ] ;
```



Diseño no distribuido…

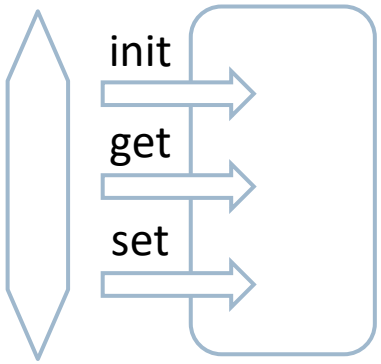



```
int      a_neltos = 0 ;
int   * a_values[100] ; // = [ [0…N1], [0…N2], … [0…NN] ] ;
char * a_keys  [100] ; // = [ "key1",    "key2",   …   "keyN" ] ;
```

```
int buscar (char *nombre)
{
    int index = -1 ;

    for (int i=0; i<a_neltos; i++)
    {
        if (!strcmp(a_keys[i], nombre)) {
            return i;
        }
    }
    return index;
}
```

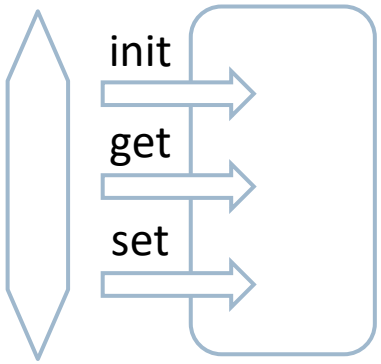
```
int insertar (char  *nombre,  int  N)
{
    a_values[a_neltos] = malloc(N*sizeof(int)) ;
    if (a_values[a_neltos] == NULL) {
        return -1 ; // en caso de error => -1
    }

    a_keys[a_neltos] = strdup(nombre) ;
    if (a_keys[a_neltos] == NULL) {
        free(a_values[a_neltos]);
        return -1 ; // en caso de error => -1
    }

    a_neltos++ ;
    return 1 ; // todo bien => devolver 1
}
```

Diseño no distribuido…




```
// Inicializar un array distribuido de N números enteros.
int init (char  *nombre,  int  N)
{
    int index = buscar(nombre) ;
    if (index != -1) return 0 ;   // Si array ya esté creado => devolver 0

    index = insertar(nombre, N) ;
    if (index == -1) return -1;  // en caso de error => -1

    return 1 ; // el array se ha creado por primera vez => devolver 1
}
```

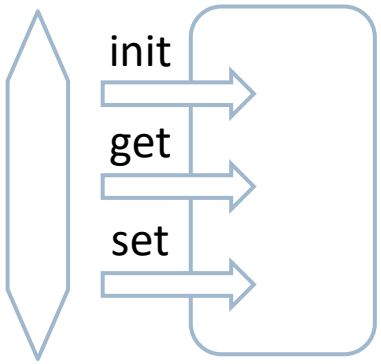
Diseño no distribuido…




```
// Inserta el valor en la posición i del array nombre.
int set (char *nombre, int i, int valor)
{
    int index = buscar(nombre) ;
    if (index == -1) return -1 ;   // Si error => devolver -1
    a_values[index][i] = valor ;
    return 1;
}

// Recuperar el valor del elemento i del array nombre.
int get (char*nombre, int i, int *valor)
{
    int index = buscar(nombre) ;
    if (index == -1) return -1 ;   // Si error => devolver -1
    *valor = a_values[index][i] ;
    return 1;
}
```

Diseño no distribuido…

algunas limitaciones



- No comprobación índice fuera de rango:
  - Guardar tamaños en init(…):

```
int      a_neltos = 0 ;
int   * a_values[100] ; // = [ [0…N1], [0…N2], … [0…NN] ] ;
char * a_keys  [100] ; // = [ "key1",     "key2",   …   "keyN" ] ;
int    * a_sizes [100] ; // = [  N1,         N2,        …   NN ] ;
```

  - Comprobar en set/get(…):

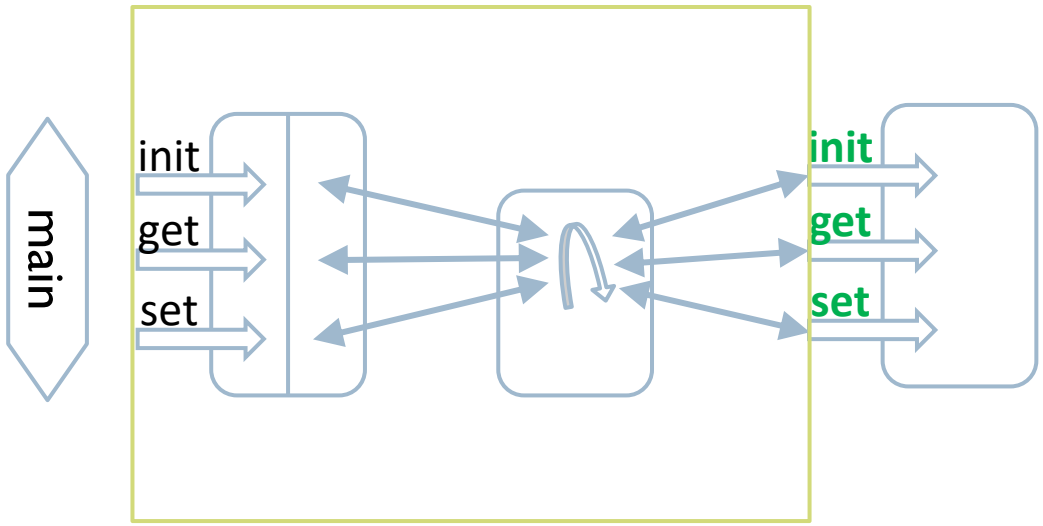
```
int      a_neltos = 0 ;
int index = buscar(nombre) ;
if (index == -1) return -1 ;
if (a_sizes[index] <= i) return -1 ;  // out-of-index
…
```

- Hay máximo de 100 vectores:
  - Usar arrays dinámicos:

```
int      a_neltos  = 0 ;
int    ** a_values = NULL ; // = [ [0…N1], [0…N2], … [0…NN] ] ;
char ** a_keys    = NULL ; // = [ "key1",     "key2",   …   "keyN" ] ;
…
a_neltos ++ ;
a_keys = realloc(a_keys, a_neltos * sizeof(char *)) ;
a_keys  [a_neltos-1] = strdup(nombre) ;
a_values = realloc(a_values, a_neltos * sizeof(char *)) ;
a_values[a_neltos-1] = malloc(N * sizeof(int)) ;
…
```

Diseño distribuido (v0.5)

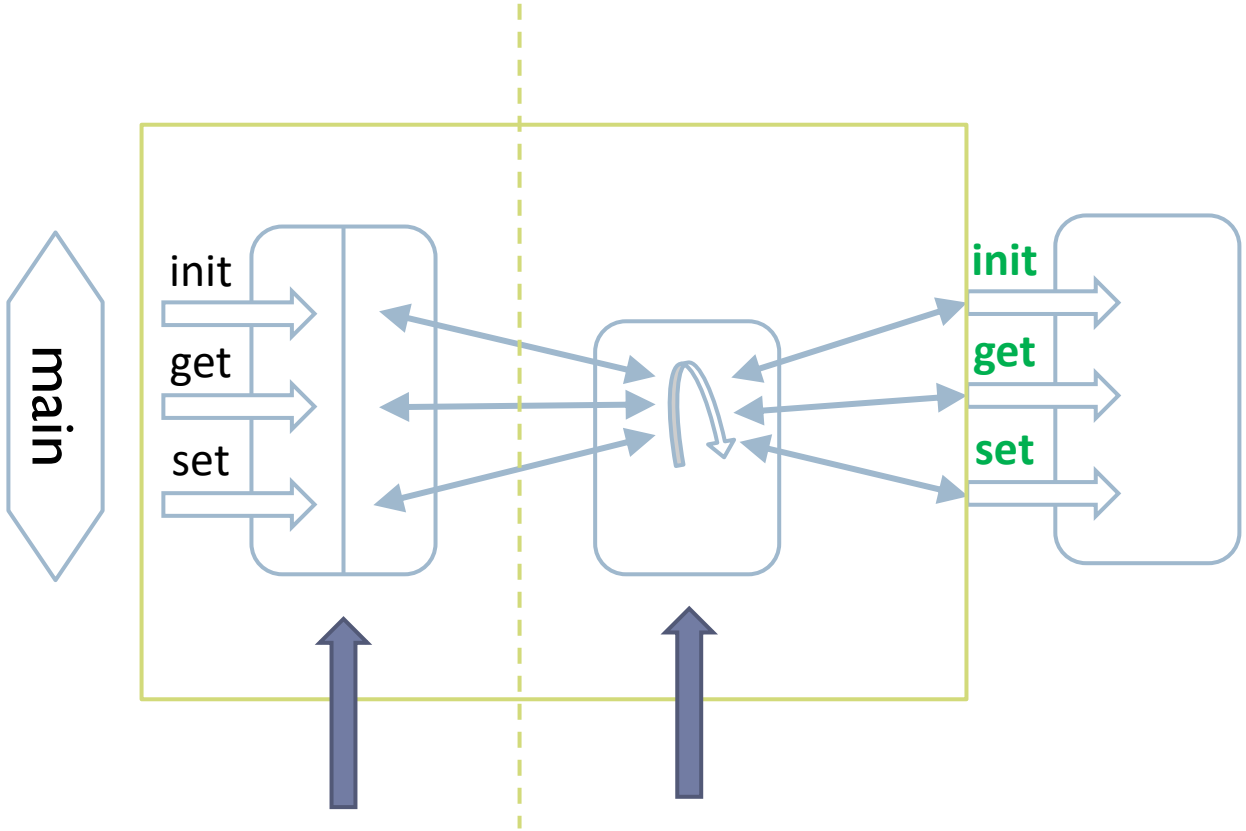
Usar un proxy para los servicios remotos



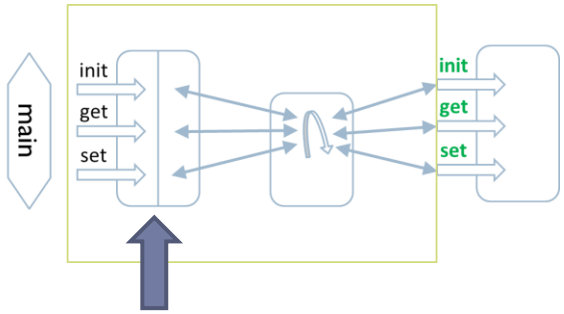
1. Sistema
2. **Distribuido**
3. Con colas de mensajes POSIX
4. Concurrentes

Diseño distribuido (v0.5)

Usar un proxy para los servicios remotos



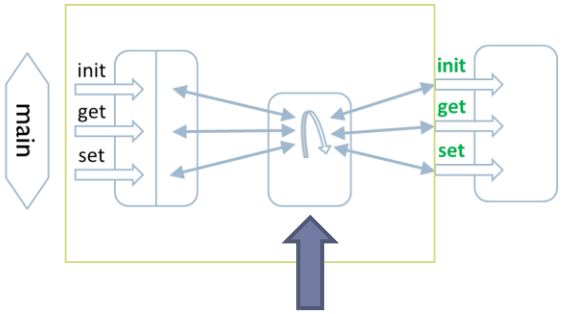
Diseño distribuido…




```
msg send_recv ( mensaje *msg ) {
    c1 = "colamsg_conectar" /SERVIDOR
    "colamsg_enviar" c1 msg
    "colamsg_recibir" c1 msg
    "colamsg_desconectar" c1
    return msg
}
```

```
int init (char *nombre, int N) {
    petición = (init, nombre, N)
    respuesta = send_recv(petición)
    return respuesta.status
}

int set (char *nombre, int i, int valor) {
    petición = (set, nombre, i, valor)
    respuesta = send_recv(petición)
    return respuesta.status
}

int get (char*nombre, int i, int *valor) {
    petición = (get, nombre, i)
    respuesta = send_recv(petición)
    *valor = respuesta.valor
    return respuesta.status
}
```

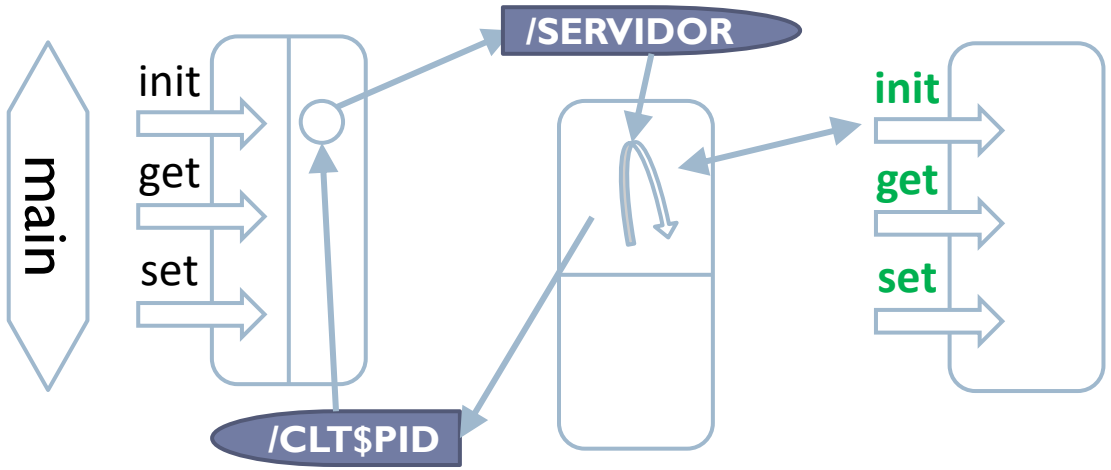
Diseño distribuido…




```
int main ( int argc, char *argv )
{
    c1 = "colamsg_crear" /SERVIDOR

    while (TRUE)
    {
        "colamsg_recibir" c1 petición
        switch( petición.operación )
        {
            case INIT:  respuesta.status = init (petición.nombre, petición.N) ;
                        break;
            case GET:   respuesta.status = get (petición.nombre, petición.i, &respuesta.valor) ;
                        break;
            case SET:   respuesta.status = set (petición.nombre, petición.i,   petición.valor) ;
                        break;
        }

        "colamsg_enviar" c1   respuesta
    }
}
```

Diseño distribuido (v0.8)

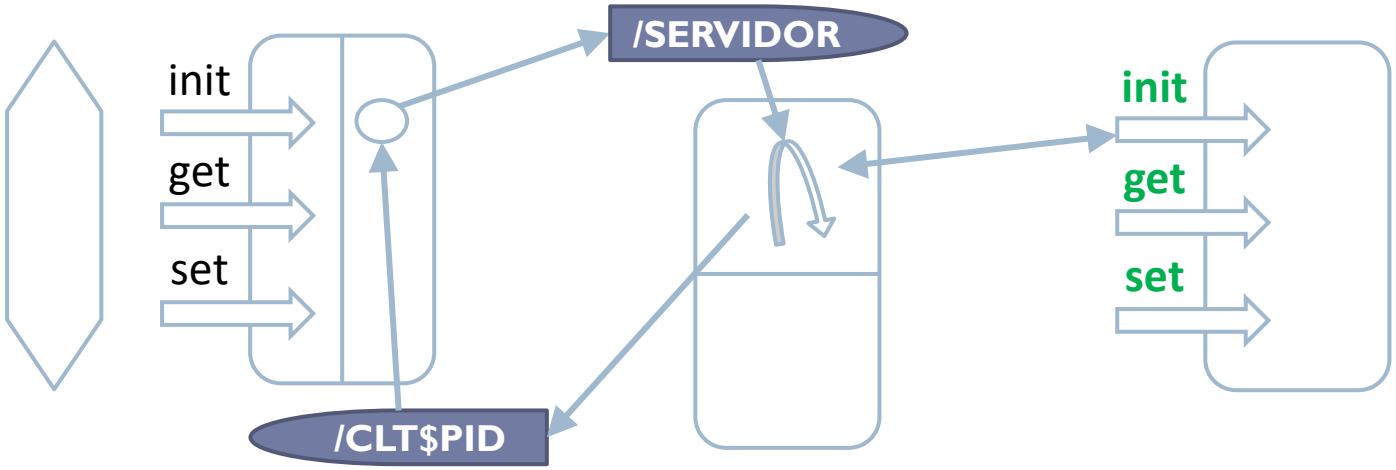
Adaptar proxy a colas de mensajes POSIX



1. Sistema
2. Distribuido
3. **Con colas de mensajes POSIX**
4. Concurrentes

Diseño distribuido (v0.8)

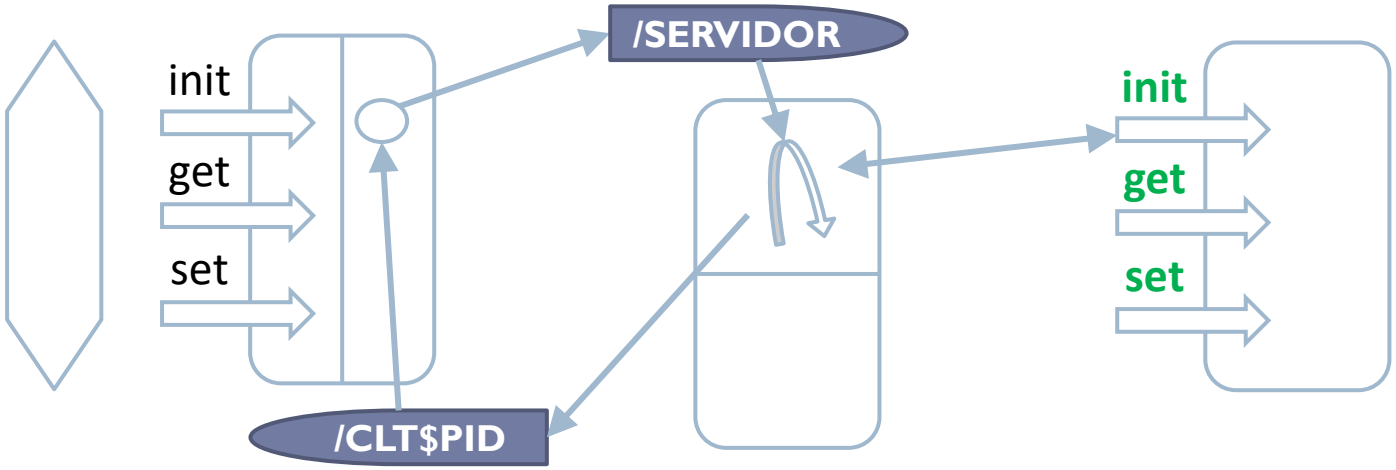
Adaptar proxy a colas de mensajes POSIX



- Las colas POSIX son unidireccionales
  - Una cola general de peticiones creada por el servidor
  - Por cada cliente activo una cola (efímera) para recibir la respuesta.
    - La cola es privada para cada el cliente con nombre único (usar getpid())

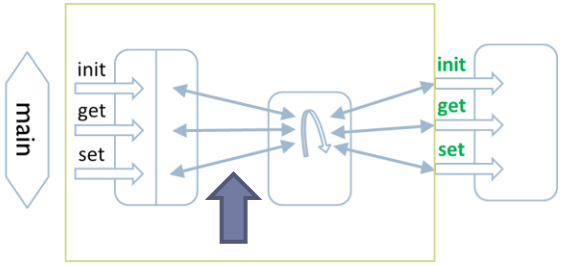
Diseño distribuido (v0.8)

Adaptar proxy a colas de mensajes POSIX



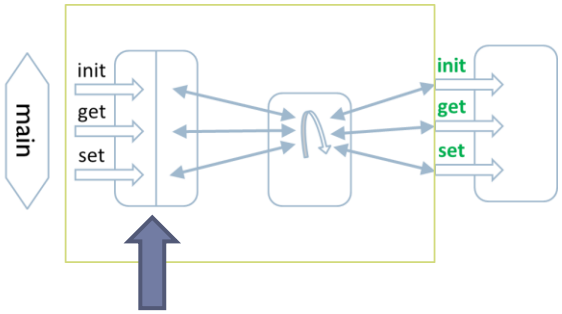
- Único formato de mensaje por cola POSIX:
  - El mensaje de petición ha de valer para todos los servicios:
    - Establecer un identificador numérico para cada servicio
    - Establecer los parámetros para cada servicio y generar una petición la fusión de todos + identificador de servicio.
  - Establecer las respuestas para cada servicio y generar una respuesta fusión.

Diseño distribuido…



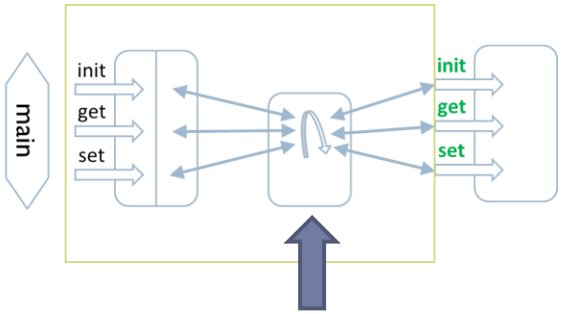

```
//  petición = op + q_name + (nombre, N) + (nombre, i, valor) + (nombre, i)
struct peticion
{
      int     op;
      char   name[MAX] ;
      int     value;
      int     i;
      char   q_name[MAX];
} ;


// respuesta = (valor, status)
struct respuesta
{
      int    value;
      char   status;
} ;
```

Diseño distribuido…



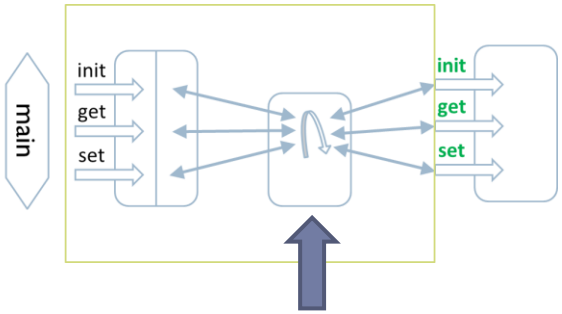

```
int get (char*nombre, int i, int *valor)
{
    struct petición p;
    struct respuesta r;
    char qr_name[1024];  unsigned int prio = 0;

    sprintf(qr_name, "%s%d", "/CLIENTE_", getpid()) ;
    int qs = mq_open("/SERVIDOR", O_CREAT|O_WRONLY, 0700, NULL) ;
    if (qs == -1) { return -1 ;  }
    int qr = mq_open(qr_name, O_CREAT|O_RDONLY) ;
    if (qr == -1) { mq_close(qs) ; return -1;  }
    p.op = 2;
    p.i = i;
    strcpy(p.nombre, nombre);
    strcpy(p.q_name, qr_name);
    mq_send    (qs,  (char *)&p,  sizeof(struct petición),    0) ;
    mq_receive(qr,   (char *)&r,  sizeof(struct respuesta),  &prio) ;
    mq_close(qs); mq_close(qr);
    mq_unlink(qr_name);
    *valor = r.value ;
    return (int)(r.status) ;
}
```

Diseño distribuido…




```
int main ( int argc, char *argv[] )
{
    struct petición p;
    unsigned int prio;

    int qs = mq_open("/SERVIDOR", O_CREAT | O_RDONLY, 0700, NULL) ;
    if (qs == -1) { return -1 ; }
    while (1)
    {
        mq_receive(qs, &p, sizeof(p), &prio) ;
        tratar_petición(&p) ;
    }
}
```

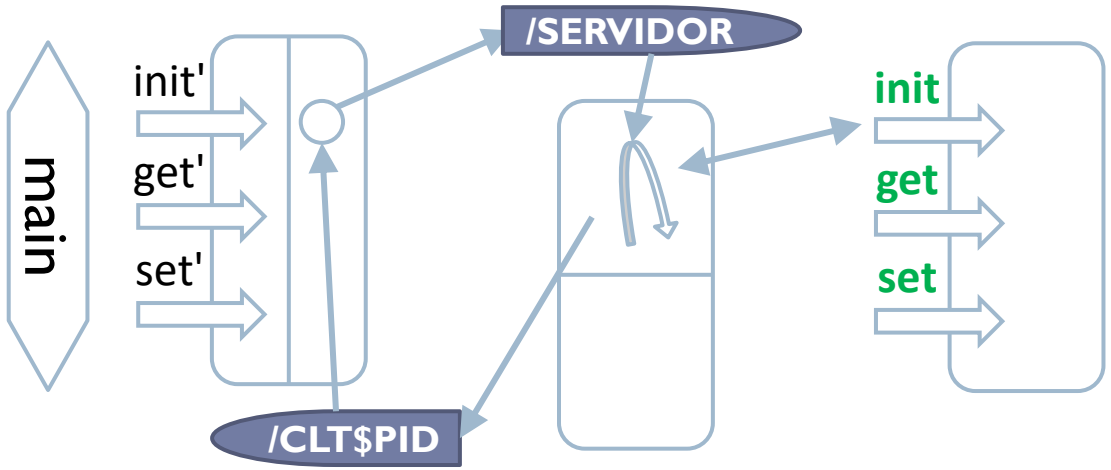
Diseño distribuido…




```
void tratar_petición ( struct petición * p )
{
    struct respuesta r ;

    switch (p->op)
    {
        case 0: // INIT
                r.status = init(p->name, p->value) ;
                break ;
        case 2: // GET
                r.status = get(p->name, p->i, &(r.value)) ;
                break ;
        case 3: // SET
                r.status = set(p->name, p->i, p->value) ;
                break ;
    }

    int qr = mq_open(p->q_name, O_CREAT|O_WRONLY, 0700, NULL) ;
    mq_send(qr, &r, sizeof(struct respuesta), 0) ;  // prio == 0
    mq_close(qr);
}
```

Diseño distribuido (v1.0)

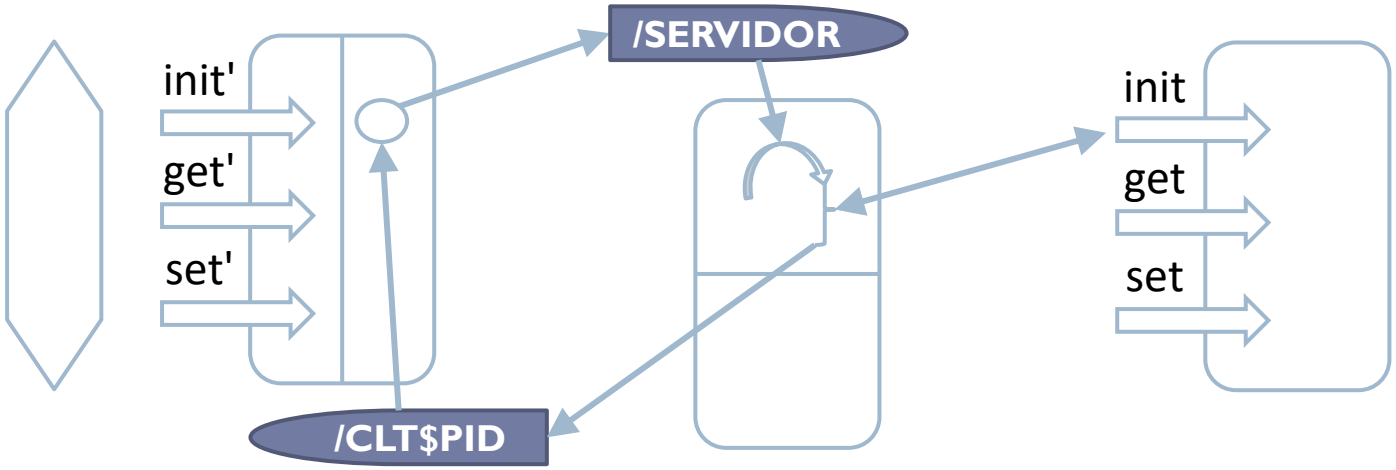
Añadir concurrencia con hilos POSIX



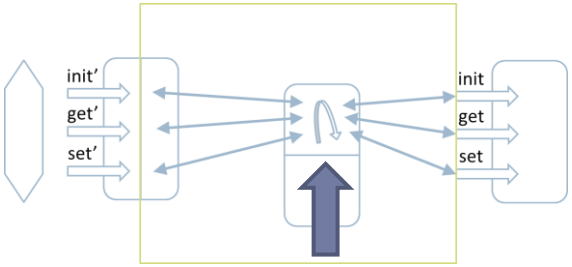
1. Sistema
2. Distribuido
3. Con colas de mensajes POSIX
4. **Concurrentes**

Diseño distribuido (v1.0)

Añadir concurrencia con hilos POSIX

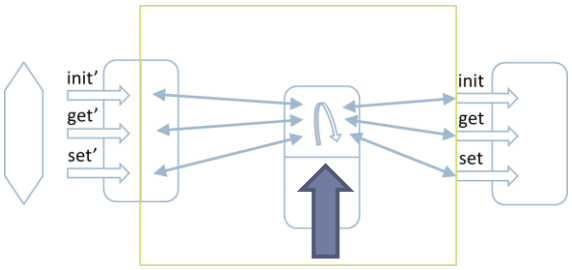


Diseño distribuido…



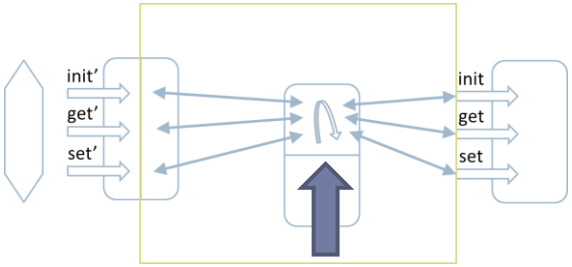

```
int main ( int argc, char *argv[] )
{
    struct petición p;
    unsigned int prio = 0;  // y algunas variables más…

    pthread_attr_init(&attr) ;
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;
    int qs = mq_open("/SERVIDOR", O_CREAT | O_RDONLY, 0700, NULL) ;
    if (qs == -1) { return -1 ; }
    while (1) {
        mq_receive(qs, &p, sizeof(struct petición), &prio) ;
        pthread_create(&thid, &attr, tratar_petición, (void *)&p) ;

        <código de espera a que se haya creado el hilo y copiado &p>
    }
}
```

Diseño distribuido…



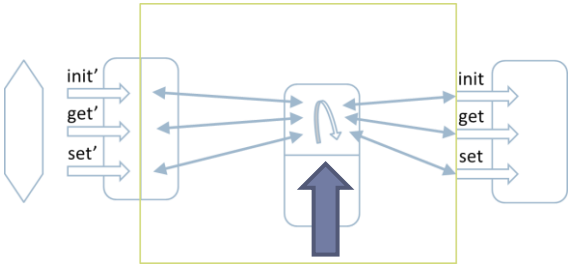

```
void tratar_petición ( struct petición * p )
{
    struct petición p_local ;
    struct respuesta r;

    <código de sincronización para "p_local = *p" y señalizar que copiado>
    switch (p->op)
    {
        case 0:   r.status = init (p->name, p.value) ;          // INIT
                  break ;
        case 2:   r.status = get (p->name, p->i, &(r.value)) ;  // GET
                  break ;
        case 3:   r.status = set (p->name, p->i, p->value) ;    // SET
                  break ;
    }

    int qr = mq_open(p->q_name, O_CREAT|O_WRONLY, 0700, NULL) ;
    mq_send(qr, &r, sizeof(struct respuesta), 0) ;  // prio == 0
    mq_close(qr);
    pthread_exit(NULL) ;
}
```

Diseño distribuido…




```
int main ( int argc, char *argv[] )
{
    struct petición p;
    struct respuesta r;
    unsigned int prio;  // y algunas variables más…

    pthread_attr_init(&attr) ;
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;
    int qs = mq_open("/SERVIDOR", O_CREAT | O_RDONLY, 0700, NULL) ;
    if (qs == -1) { return -1 ; }
    while (1) {
        mq_receive(qs, &p, sizeof(struct petición), &prio) ;
        pthread_create(&thid, &attr, tratar_petición, (void *)&p) ;

        pthread_mutex_lock(&sync_mutex) ;
        while (sync_copied == FALSE) {
                pthread_cond_wait(&sync_cond, &sync_mutex) ;
        }
        sync_copied = FALSE ;
        pthread_mutex_unlock(&sync_mutex) ;
    }
}
```

Diseño distribuido…




```
void tratar_petición ( struct petición * p )
{
    struct petición p_local ; struct respuesta r; unsigned prio = 0;

    pthread_mutex_lock(&sync_mutex) ;
    p_local = *p ;
    sync_copied = TRUE ;
    pthread_cond_signal(&sync_cond) ;
    pthread_mutex_unlock(&sync_mutex) ;

    switch (p.op)
    {
      case 0:   r.status = init (p.nombre, p.ni) ;
                break ;
      case 2:   r.status = get (p.nombre, p.i, &(r.valor)) ;
                break ;
      case 3:   r.status = set (p.nombre, p.i, p.valor) ;
                break ;
    }

    int qr = mq_open(p->q_name, O_CREAT|O_WRONLY, 0700, NULL) ;
    mq_send(qr, &r, sizeof(struct respuesta), prio) ;
    mq_close(qr);
    pthread_exit(NULL) ;
}
```

Grupo ARCOS
Universidad Carlos III de Madrid

Lección 3
Ejercicios de paso de mensajes

Sistemas Distribuidos
Grado en Ingeniería Informática