Concurrent programming

Inter-process IPC communication

IPC

There are 3 types of inter-process communication in system V.

- message queue
- semaphores
- shared memory

IPC

	Message queue	Semaphore	Shared memory
header file	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
create or open system function	msgget	semget	shmget
control operation system function	msgctl	semctl	shmctl
transfer system functions	msgsnd msgrcv	semop	shmat shmdt

IPC

The kernel holds information about each interprocess communication channel in the structure:

```
#include <sys/ipc.h>
struct ipc_perm
{
ushort uid; /* owner user id */
ushort gid; /* owner group id */
ushort cuid; /* creator user id */
ushort cgid; /* creator group id */
ushort mode; /* access mode */
ushort seq; /* sequential number */
key_t key; /* key */
}
```

IPC

- Functions are used to get the value of this structure:

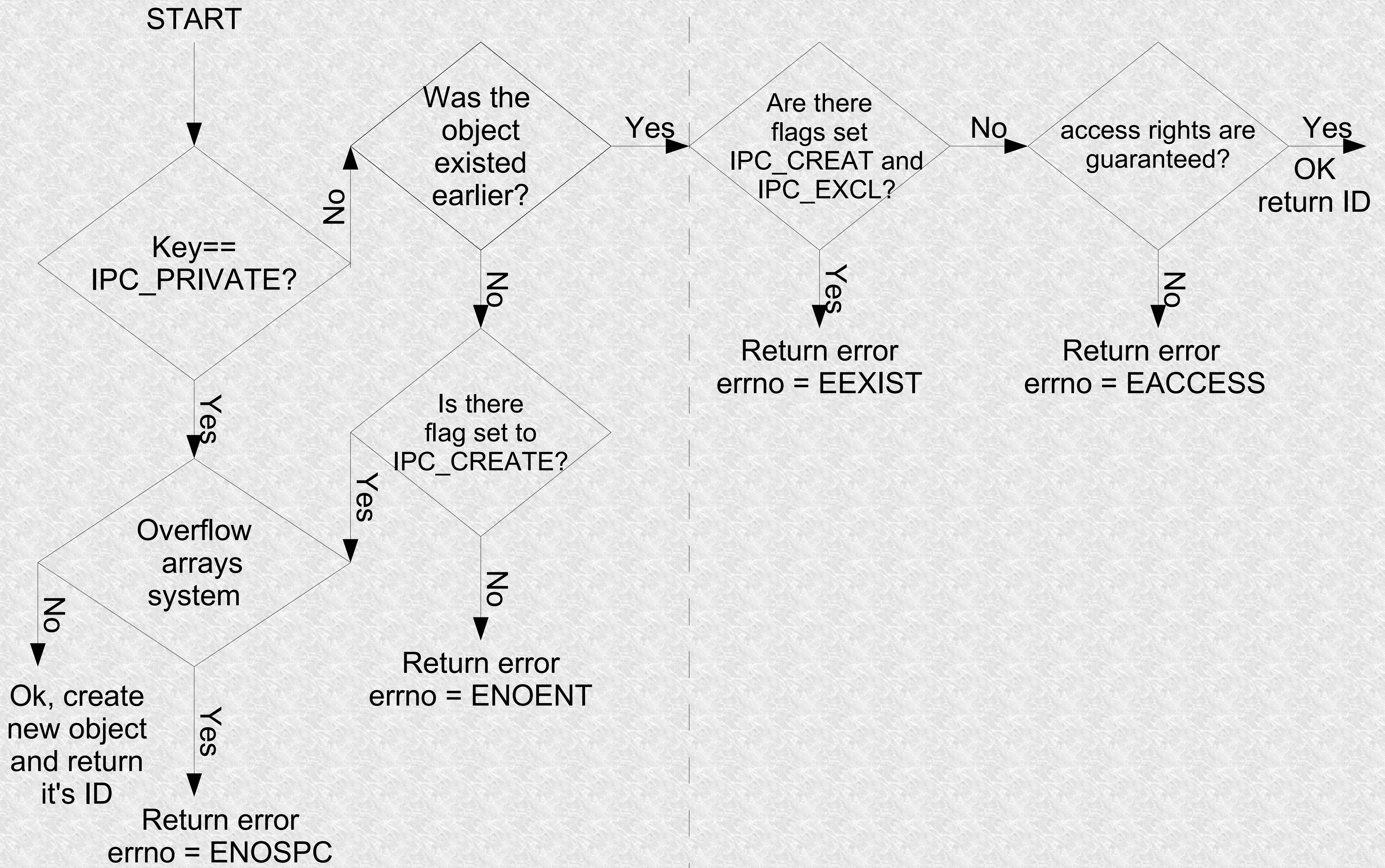
  msgctl, **semctl** and **shmctl**

- All three functions for creating an IPC communication channel **msgget**, **semget**, and **shmget** take the key_t key and return the channel ID..

IPC

They all also have the flag argument which

- The least significant 9 bits specify the channel access mode
- the rest define whether a channel should be created or not, etc..
  - if the key argument is **IPC_PRIVATE**, a private interprocess communication channel will be created.
  - There is no such combination of path and id that ftok would generate **IPC_PRIVATE**
  - **IPC_CREAT** in the flag argument will create a new element in the kernel system table, if it is missing, it will be passed as a return value if there is one.
  - if the flag has **IPC_CREAT + IPC_EXCL**, the channel will be created only if it was not there, if it was, the function will return an error.
  - **IPC_EXCL** has no effect without **IPC_CREAT**



Create new object | Use existing object

IPC

- during creation initializes the ipc_perm field the lowest 9 bits of the flag argument
- cuid, cgid, uid and gid takes the valid user and group identifiers for the calling process.
- channel creator IDs never change (cuid and cgid)
- Owner IDs can be changed by calling msgctl, semctl or shmctl appropriately
- *ctl are also used to change access rights
- each operation on IPC channels (write, read) causes checking similar to file system access rights checking

IPC

- Checking is based on the **ipc_perm** field
  - The supervisor process is always granted access rights
  - if the uid or cuid and the appropriate access bit are correct, the process is granted access.
  - if the gid or cgid match the corresponding permission bit, the process is granted permission
  - if the above fails, the appropriate bit of access rights for others must be set

IPC Message Queues

- all messages are stored in the kernel and assigned a **message queue identifier**
- msqid – identifies a specific message queue
- processes read and write to any message queue
- One process is not required to wait for the message before the other one starts writing
- the process may put the message on the queue and exit.
- each message in the queue has the following attributes
  - **type** – a long integer
  - **length** - data portion, it can be 0
  - **data** - if the length is greater than 0

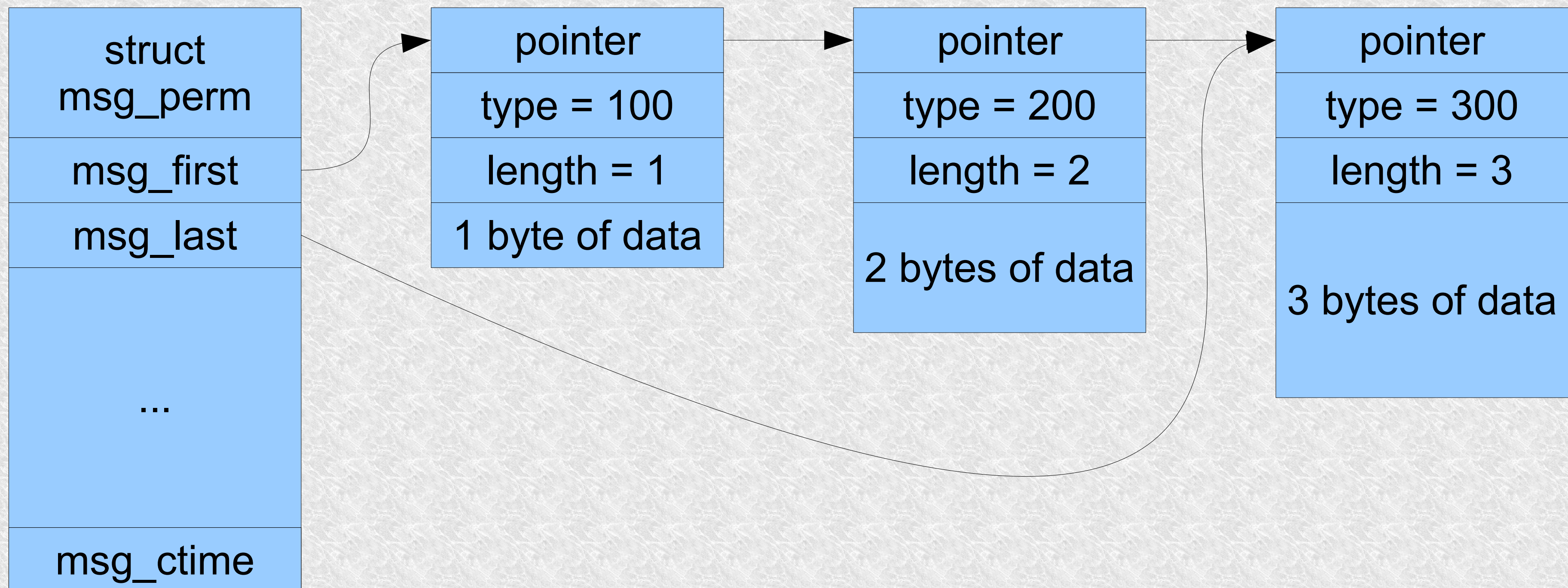
IPC Message Queues

For the message queue, the kernel maintains this structure

```
#include <sys/types.h>
#include <sys/ipc.h>
struct msqid_ds.
{
struct ipc_perms msg_perms; /* access rights structure for an
operation */
struct msg *msg_first; /* pointer to the first message in the queue
*/
struct msg *msg_last; /* pointer to the last message in the queue */
ushort msg cbytes; /* the current number of bytes in the queue */
ushort msg_gnum; /* the current number of messages in the queue */
ushort msg_gbytes; /* maximum number of bytes for the queue */
ushort msg_lspid; /* id of the process that last wrote to the queue
*/
ushort msg_lrpid; /* ID of the process that last called msgrcv */
time_t msg_strime; /* time of the last call to msgsnd */
time_t msg_rtime; /* time of the last call to msgrcv */
time_t msg_ctime /* time of the last call to msgctl which changed
the value of the above fields */
};
```

IPC Message Queues

Suppose we have three messages in the queue with a length of 1,2,3 bytes of types 100,200,300 and in this order it came:



IPC Message Queues

Msgget is used for creation

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

   int msgget (key_t key, int msgflag);
```

the function will return the queue id or –1 if it fails

IPC Message Queues

msgflag defines the access rights:

Numerical value	symbolic constant	meaning
0400	MSG_R	reading for the owner
0200	MSG_W	writing for the owner
0040	MSG_R >> 3	reading for the group
0020	MSG_W >> 3	writing for the group
0004	MSG_R >> 6	reading for others
0002	MSG_W >> 6	writing for others
1000	IPC_CREAT	
2000	IPC_EXCL	
4000	IPC_NOWAIT	

IPC Message Queues

It is used to put a message on the queue:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr, int
length, int flag);
```

- lenhgth specifies the length of the message in bytes it is the number of user-defined data bytes, i.e. those after the long field, it can also be 0.
- flag can take the symbolic constant IPC_NOWAIT or 0.
- if there is IPC_NOWAIT then if the queue is full or there are too many messages across the system the function will return –1 immediately and errno = EAGAIN.
- when msgsnd is successful, we get 0 return.

IPC Message Queues

- ptr is a pointer to a structure with the following pattern

```
struct msgbuf
{
long mtype; /* message type */
char *mtext; /* message data */
}
```

- mtext is confusing because the data may be different
- by pattern we mean that ptr must point to an integer of long type which contains the message type and precedes the message itself (if length> 0)
- the kernel does not interpret the message body
- you can define your own structure, e.g..

```
typedef struct my_msgbuf {
    long mtype;
    short mshort; /* enything */
    char mchar[8]; /* whatever */
} Message;
```

IPC Message Queues

For pickup is used

```
int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype, int flag);
```

- ptr indicates where the message should be written
- length – ptr data size
- msgtype – specifies the message desired by the recipient
  - 0 marks the first one to enter the queue (FIFO rule)
  - >0 the first message of the same type
  - <0 the first message of the smallest types that are not greater than the absolute value of msgtype.

IPC Message Queues

For an exemplary queue with types 100L, 200L and 300L, we have

Argument msgtype	The type of the message passed
0L	100L
100L	100L
200L	200L
300L	300L
-100L	100L
-200L	100L
-300L	100L

IPC Message Queues

- flag – specifies what to do when there are no messages in the queue
  - when IPC_NOWAIT is set then immediately return, the function returns -1 and errno = ENOMSG
  - if there is no IPC_NOWAIT, the process waits for one of 3 events to occur
    - you will receive a message of the requested type
    - the message queue is removed from the system.
    - the process will catch the appropriate signal
  - if flag has MSG_NOERROR then if we get more data than indicated by length, the excess will be skipped and the function will not fail

IPC Message Queues

It is used to control the message queue

```
int msgctl(int msqid, int cmd, struct msqid_ds.
*buff);
```

where cmd is:

IPC_RMID – 0 remove the queue
IPC_SET – 1 sets on base the buff
IPC_STAT – 2 gets to the buff

IPC Message Queues

Client – Server.

```
/* msgq.h */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/errno.h>

extern int errno;
#define MKEY1 1234L
#define MKEY2 2345L
#define PERMS 0666
```

IPC Message Queues

```
#include „msgq.h”

main() //for server
{
int readid, writeid;
if ((readid = msgget(MKEY1,PERMS|IPC_CEAT))<0)
   perror(„server: I can't open the queue 1”);
if ((writeid = msgget(MKEY2, PERMS | IPC_CREAT)) <0)
   perror(„server: I can't open the queue 2”);
server(readid, writeid);
exit(0);
}
```

IPC Message Queues

```
#include „msgq.h”

main()//for client
{
int readid, writeid;
if ((writeid = msgget(MKEY1,0))<0)
   perror(„client: I can't open the queue 1”);
if ((readid = msgget(MKEY2,0)) < 0)
   perror(„client: I can't open the queue 2”);
client(readid, writeid);
/* removing queue */
if (msgctl(readid, IPC_RMID,(struct msqid_ds*)0) <0)
   perror(„client: I can't remove the queue 1”);
if (msgctl(writeid,IPC_RMID, NULL) < 0)
   perror(„client: I can't remove the queue 2”);
exit(0);
}
```

IPC Message Queues

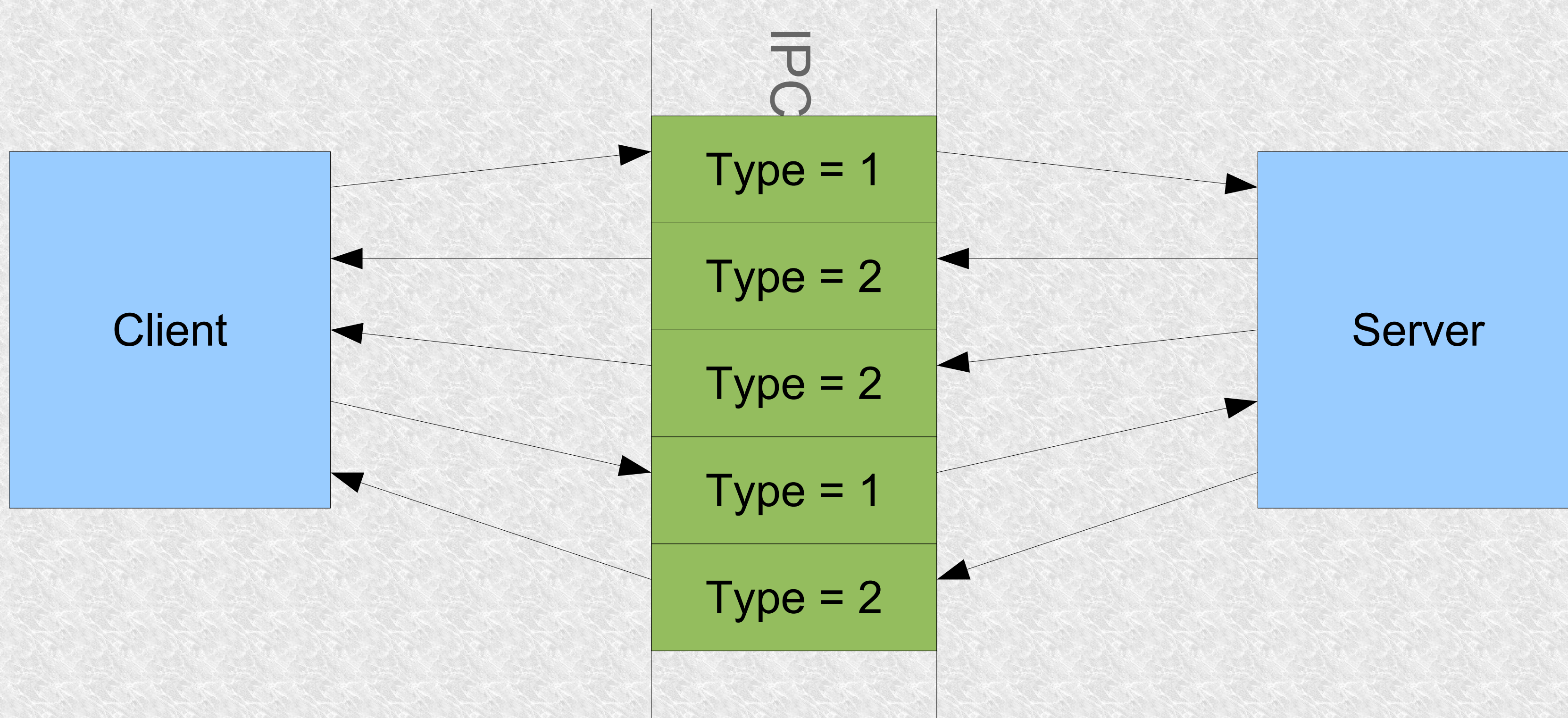
```
#include „mesgq.h”
void mesg_send(int id, Mesg *mesgptr)
{
if (msgsnd(id, (char*)&(mesgptr->mesg_type),
   mesgptr->mesg_len,0) !=0)
      perror(„error sending the message”);
}

int mesg_recv(int id, Mesg *mesgptr)
{int n;
n = msgrcv(id, (char*) &mesgptr->mesg_type),
   MAXMESGDATA, mesgptr->mesg_type, 0);
if ((mesgptr->mesg_len = n) < 0)
   perror(„receive error”);
return(n); /* if end of file is n = 0 */
}
```

IPC Message Queues

Two-way communication can be achieved by using message types

- type 2 means messages from server to client
- type 1 means messages from client to server



IPC Message Queues

By using message types, two-way communication for the server and multiple clients can be achieved.

- When sending a message, the client sets the type = 1 and places its pid in the content
- The server receives all messages where type = 1
- Reads the sender's pid from the content
- It responds by putting a message into a queue where type = pid
- The client reads all messages with a type like its pid.

IPC Message Queues