Software Defined Networking: Distributed Systems and Trust Computation

Overview

- The lack of support to distributed network in Openflow standard.
- Innovative way to interconnect multiple controllers.
- Acts like a messenger system between controllers.
- Controllers pose queries of certain network node addresses.
- Replies are sent by corresponding controllers that own those nodes.
- Flexible library is provided to make the controller docile to relay protocol.
- Single entry, multi process and single exit structure.

The Relay Structures

struct controller

- int id
- int bcast_sock
- int sock
- struct sockaddr_in addr

struct bcast_msg_struct

- int id
- int done
- char *msg
- struct controller *sender

union node

- int tag
- struct controller *ctrlr
- struct bcast_msg_node *bmn
- union node *nxt
- union node *prev

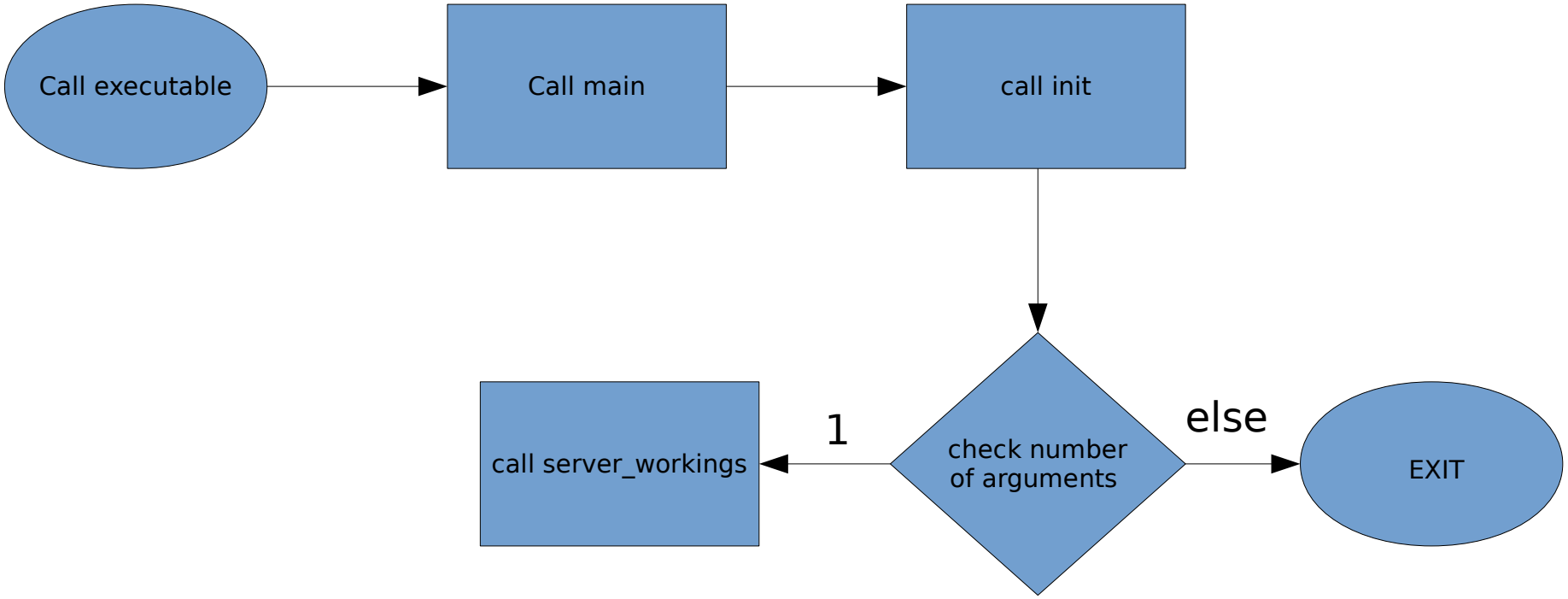
struct broadcast_struct

- struct controller *cli
- char *cmds

The Relay Modules

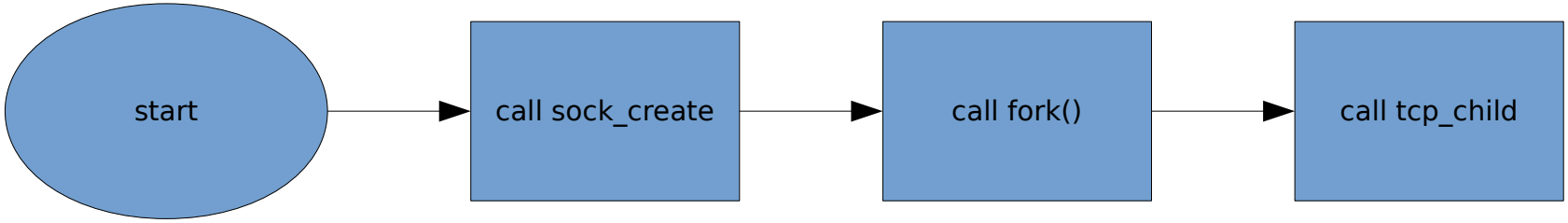
main.c

- int init(int argc)
- int main(int argc, char *argv[])



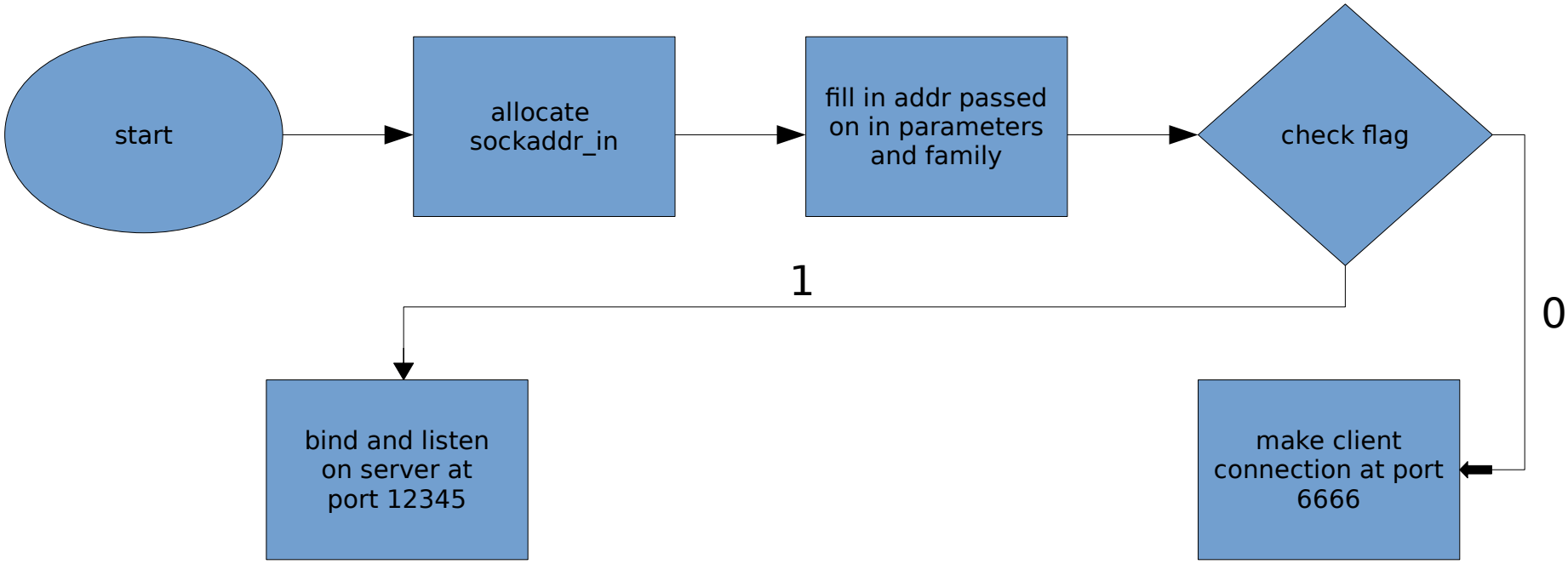
server.h

- int server_workings(char *argv)



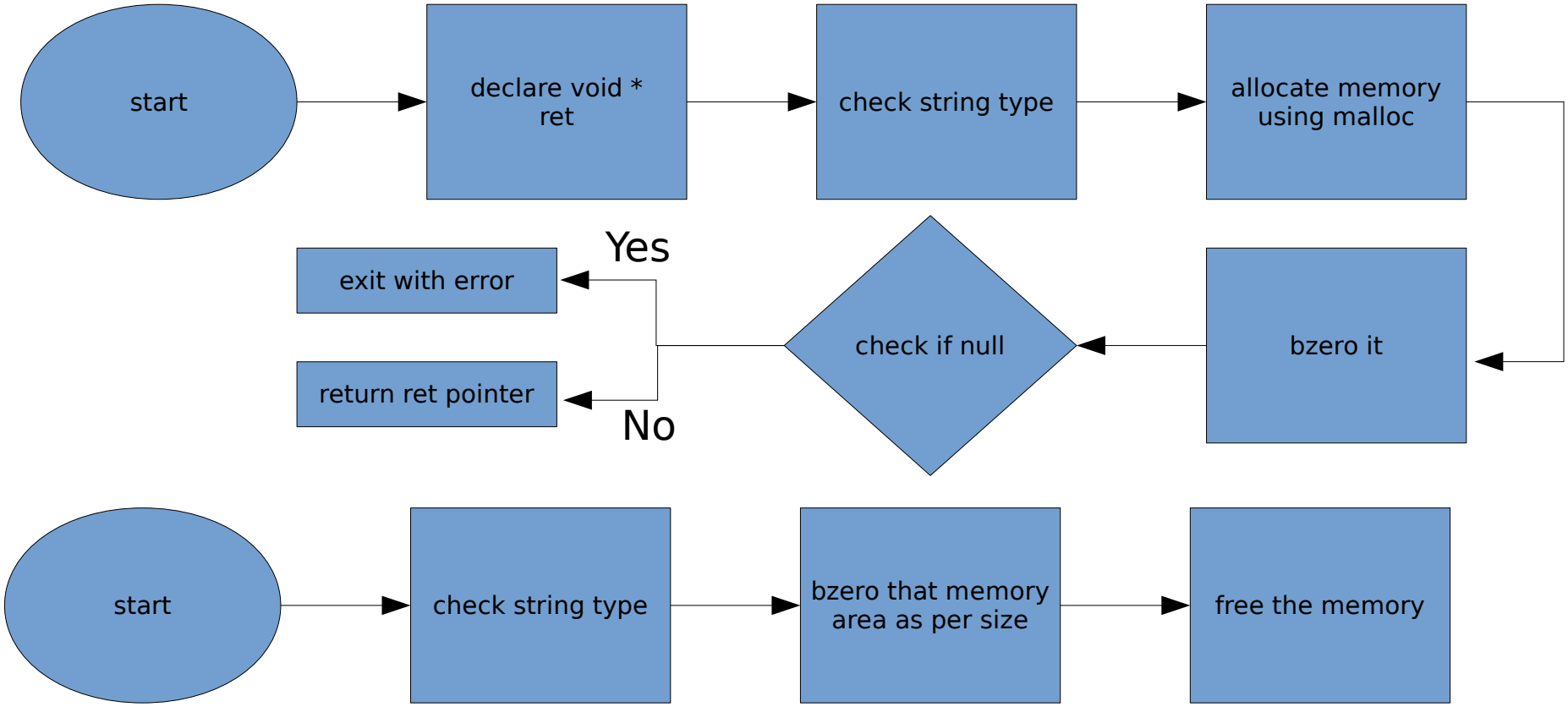
sock_create.h

- int sock_create(char *addr, int flag)



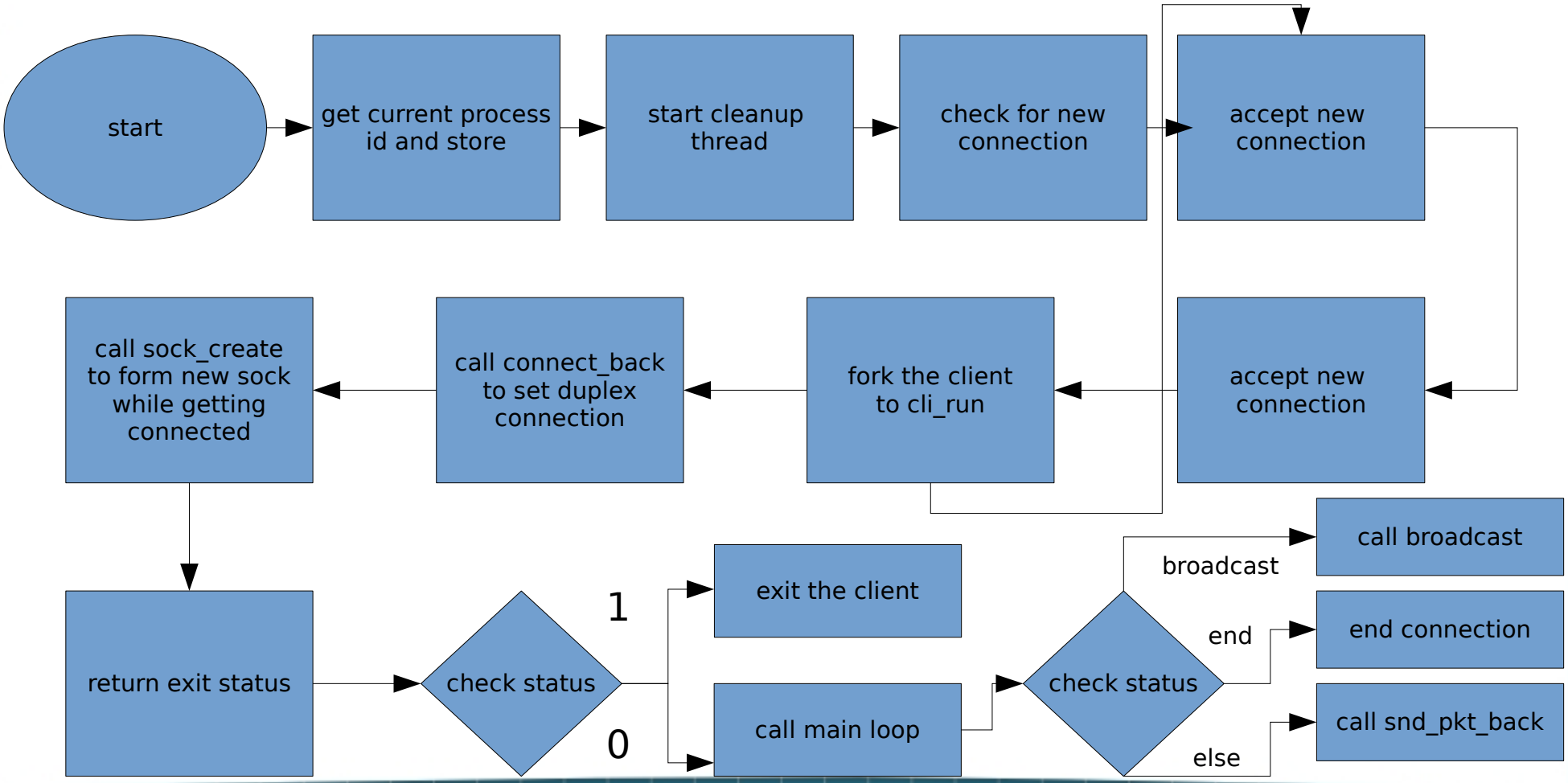
allocate.h

- void *allocate(char *type, int size)
- void deallocate(void *a, char *type, int size)



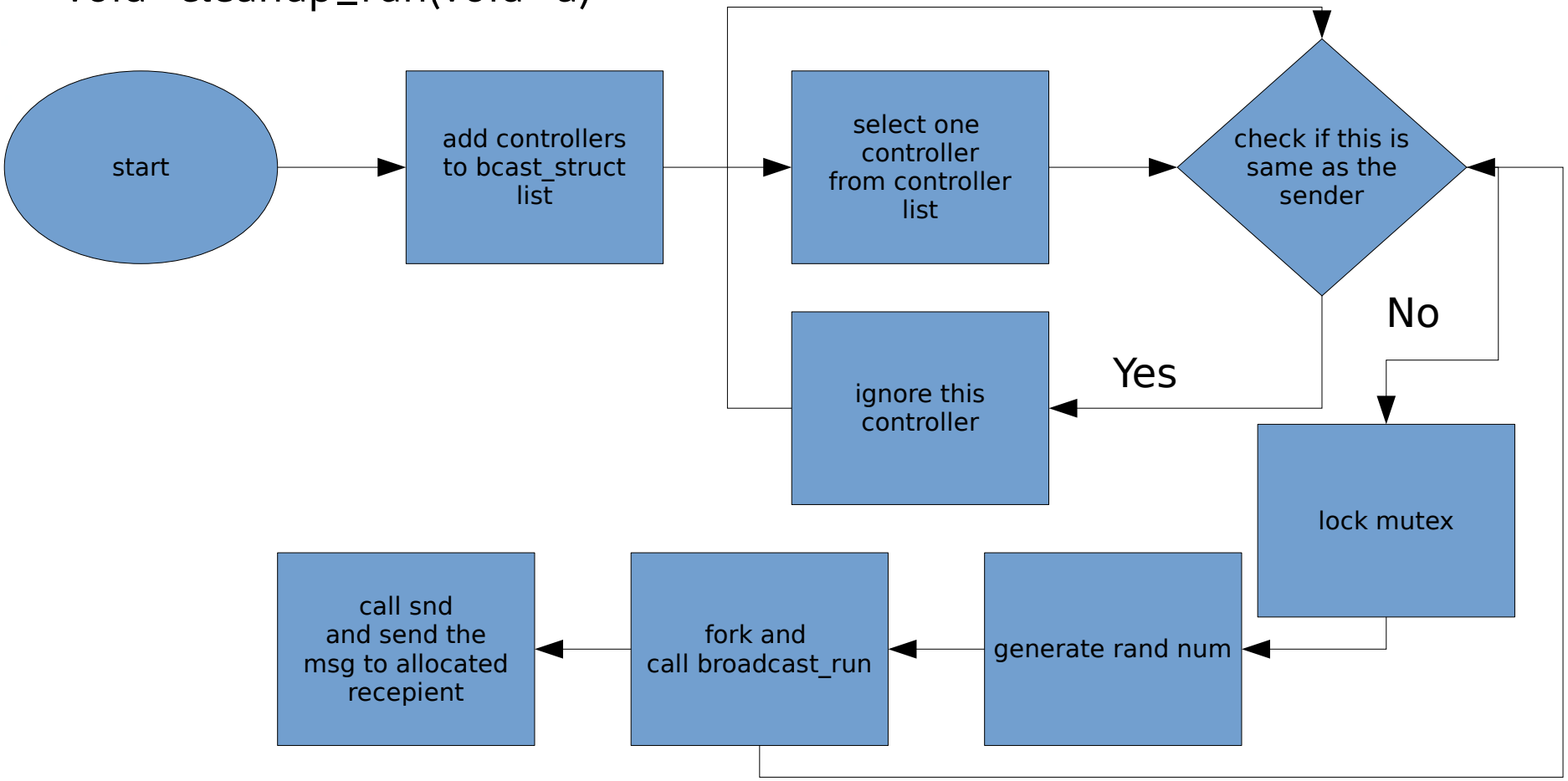
tcp_child.h

- void tcp_child()
- void cli_run(union node *client)
- int connect_back(struct controller *sender)

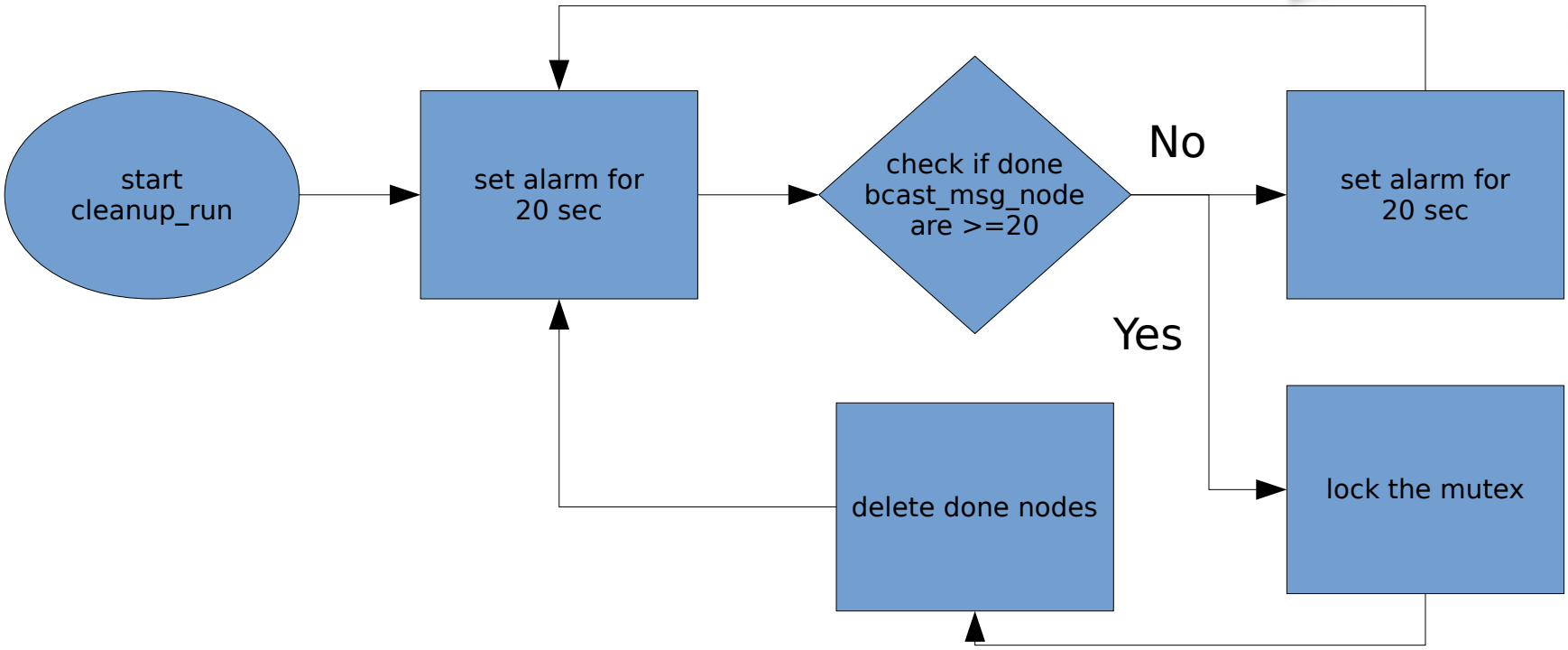


broadcast.h

- int broadcast(struct controller *sender, char *cmds)
- void broadcast_run(struct broadcast_struct *b_struct)
- void *cleanup_run(void *a)

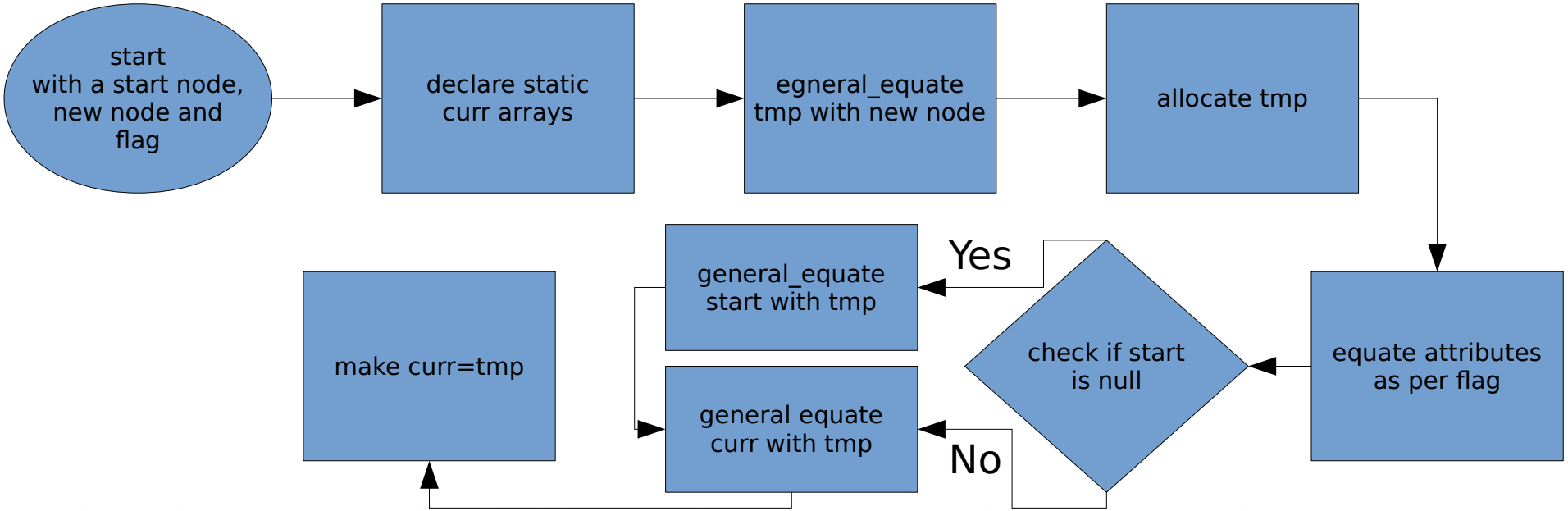


broadcast.h

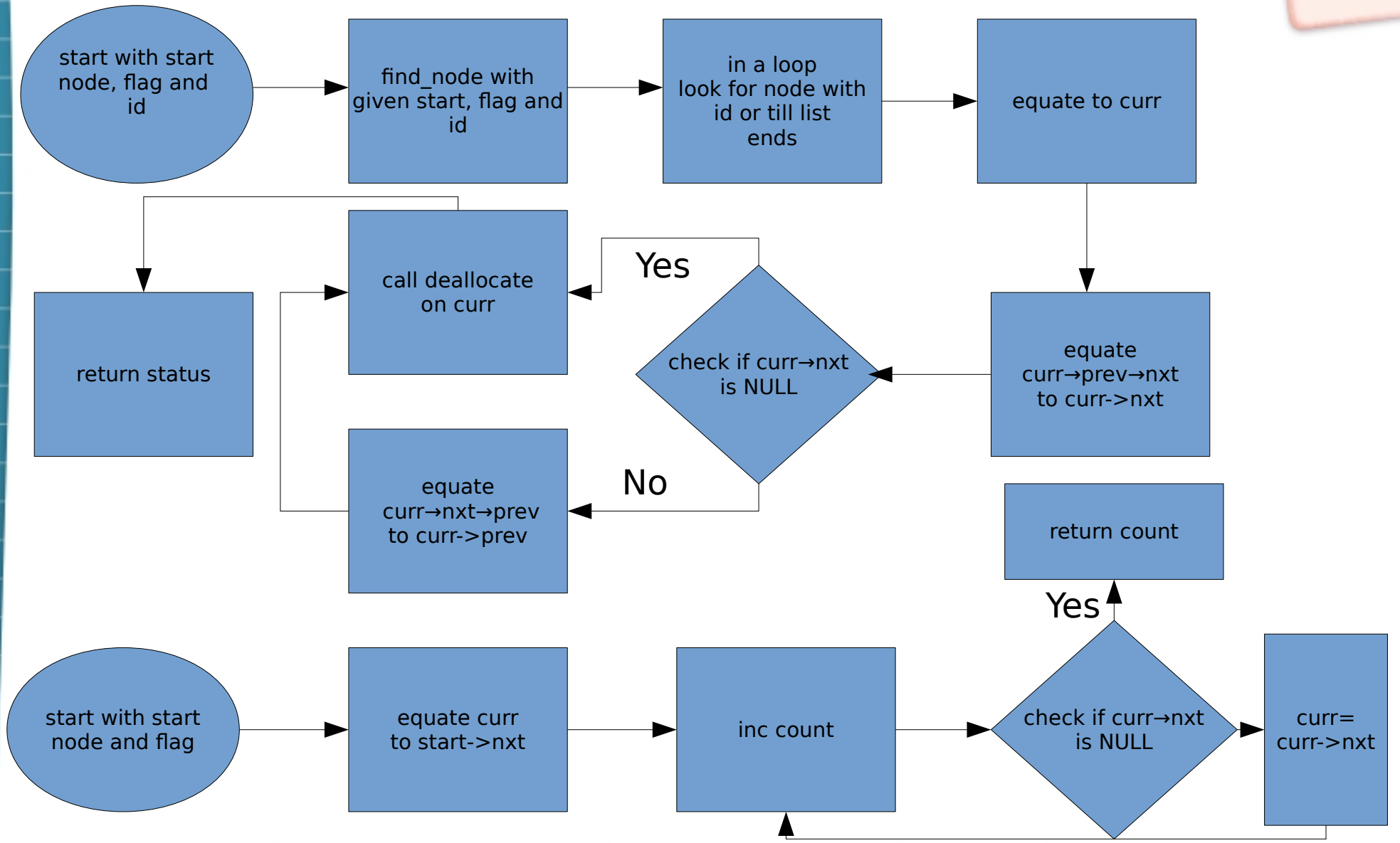

start
cleanup_run
set alarm for
20 sec
check if done
bcast_msg_node
are >=20
No
set alarm for
20 sec
Yes
lock the mutex
delete done nodes

list.h

- void add_node(union node *new, union node *start, int flag)
- void general_equate(union node *a, union node *b, int flag)
- union node *find_node(union node *start, int tag)
- int del_node(union node *start, int flag, int tag)
- int list_len(union node *start)

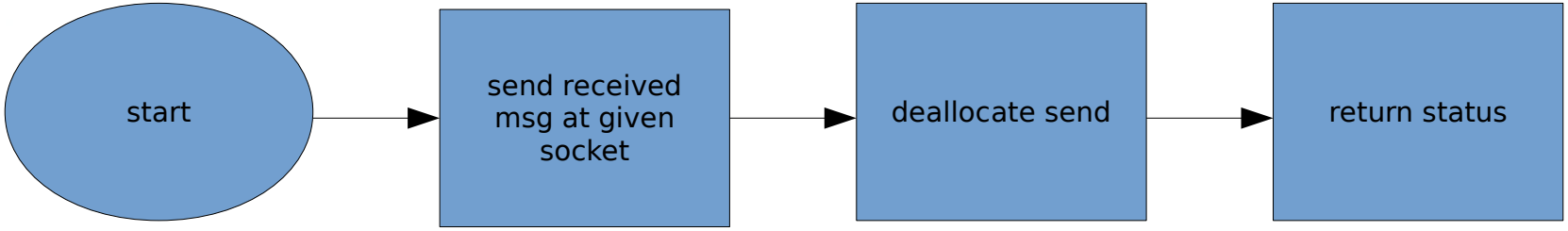


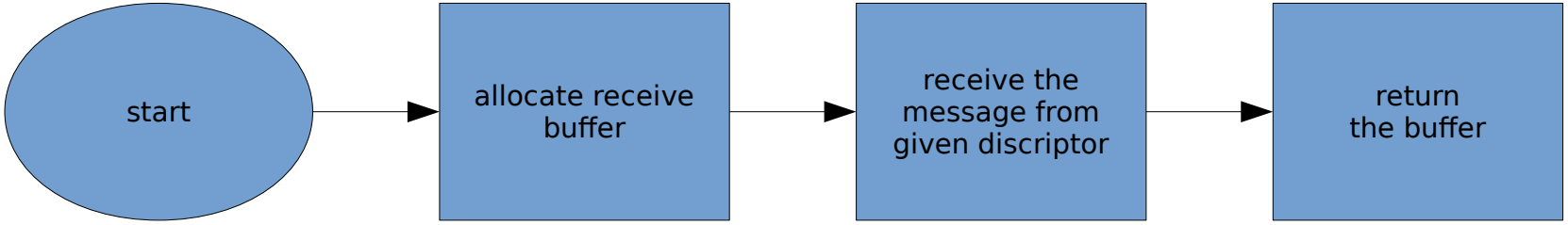
list.h


start with start node, flag and id
find_node with given start, flag and id
in a loop look for node with id or till list ends
equate to curr
return status
call deallocate on curr
Yes
check if curr→nxt is NULL
equate curr→prev→nxt to curr->nxt
equate curr→nxt→prev to curr->prev
No
return count
Yes
start with start node and flag
equate curr to start->nxt
inc count
check if curr→nxt is NULL
curr= curr->nxt

snd_rcv.h

- int snd(struct controller *cli, char *cmds, char *reason, char *retval, int free_it)
- char *rcv(struct controller *cli, int sock, char *reason, char *retval)





The Relay bottle-neck reduction methods

The fork() function

- #include<sys/types.h>
- #include<unistd.h>
- pid_t fork(void)
- Returns 0 in child and Process ID of child in parent.
- Creates a new process by duplicating a new process.

Multi Process v/s Multi thread

- Sub processes have new memory mappings.
- Sub process memory dies when the process dies.
- Threads have save memory maps as the parent process
- Threads memory is safe on their termination.

Where we use sub-processes?

- The tcp server is a sub process.
- Each new client is a sub process.
- Each broadcast call for every client is a sub process.

Where we use threads?

- The cleanup module to clean the done with broadcast structures.
- The alarm module is a alarm set in separate thread.

Why do we use sub-process?

- The tcp_child server as sub process can be monitored and killed of by calling parent process in event of mishandling.
- Each client has OS level compartmentalization to guarantee security over memory sharing and buffer overflow attacks.
- The broadcast calls are gracefully spread over all cores for a real simultaneous transaction.

Why we use threads?

- The cleanup module must be a part of main tcp server’s memory yet run autonomously.
- The alarm is a short time keeping module to trigger cleanup which again needs to save same memory.

The Synchronization

- The greatest part of pthread.h
- pthread_mutex_t is PTHREAD_MUTEX_INITIALIZER here.
- int pthread_mutex_lock(pthread_mutex_t *)
- int pthread_mutex_trylock(pthread_mutex_t *)
- int pthread_mutex_unlock(pthread_mutex_t *)

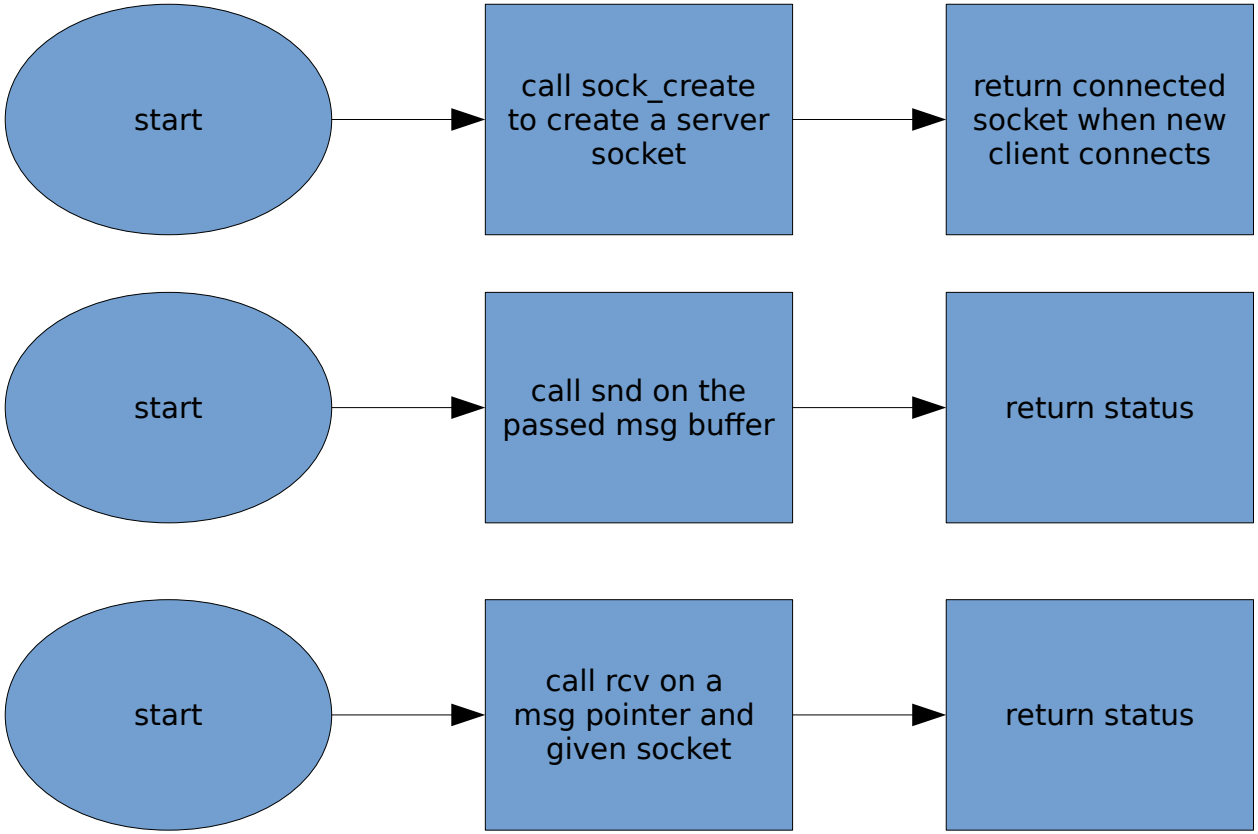
Our implementation

- We use a pthread_mutex_t speaker = PTHREAD_MUTEX_INITIALIZER initialized with common memory in global_defs.h
- This keeps all the client nodes in check while being broadcasted to by the broadcast sub-processed or cleaned up by cleanup module thread.

The Controller Modules

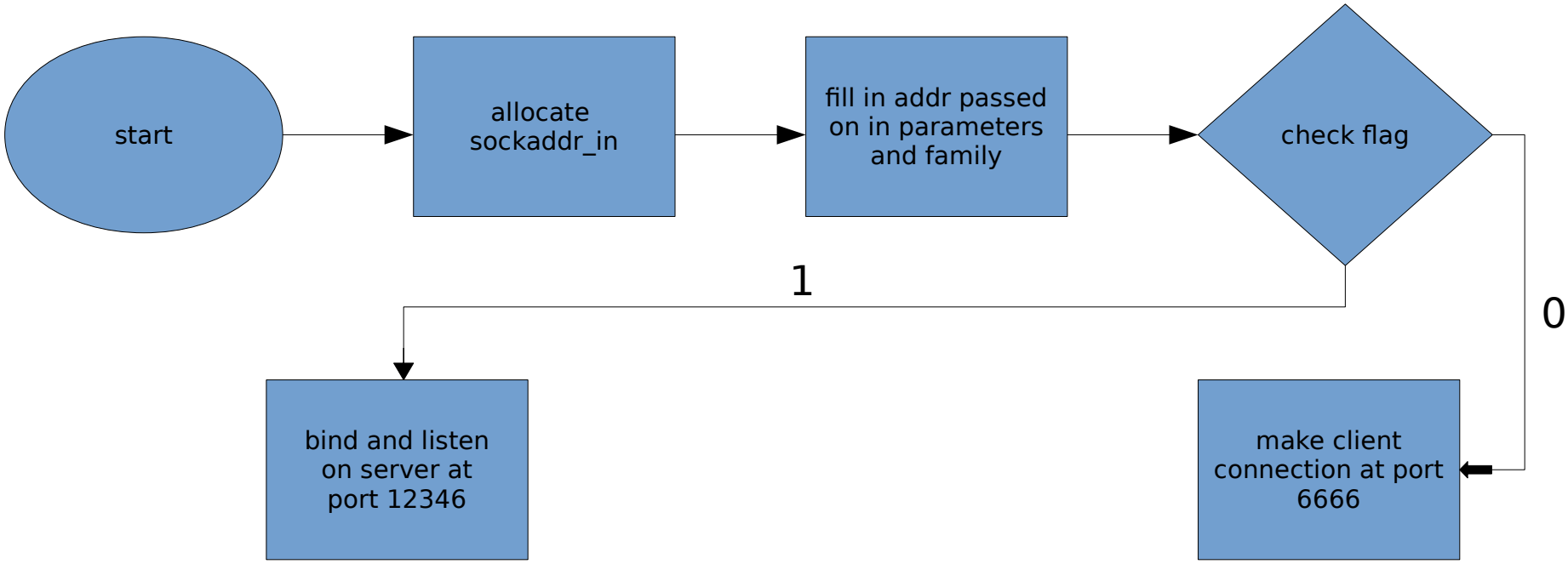
tcp_connector.h

- int get_connection_back(int sock)
- int send_to_relay(int sock, int flag, char *addr)
- char *rcv_bcast(int sock)



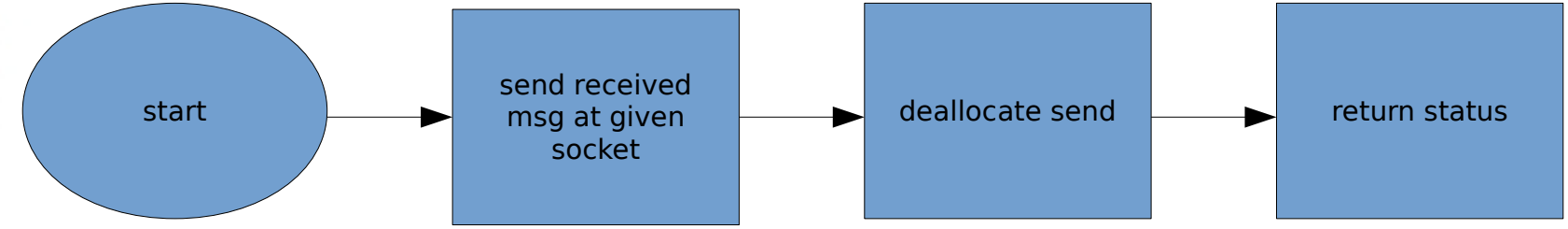
sock_create.h

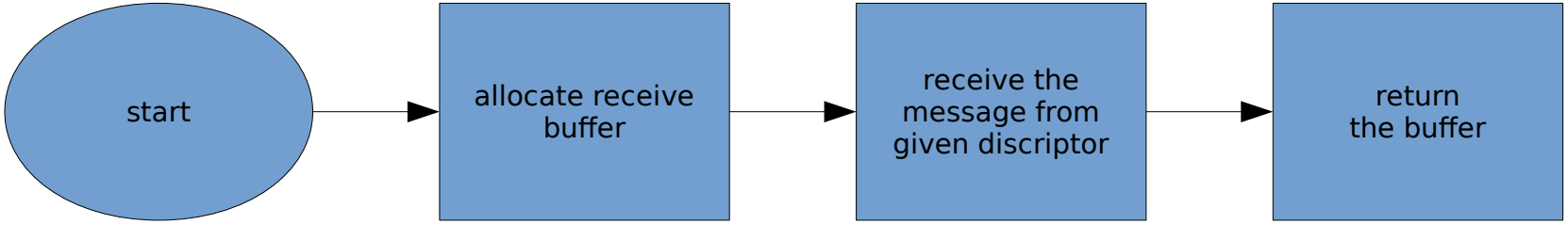
- int sock_create(char *addr, int flag)



snd_rcv.h

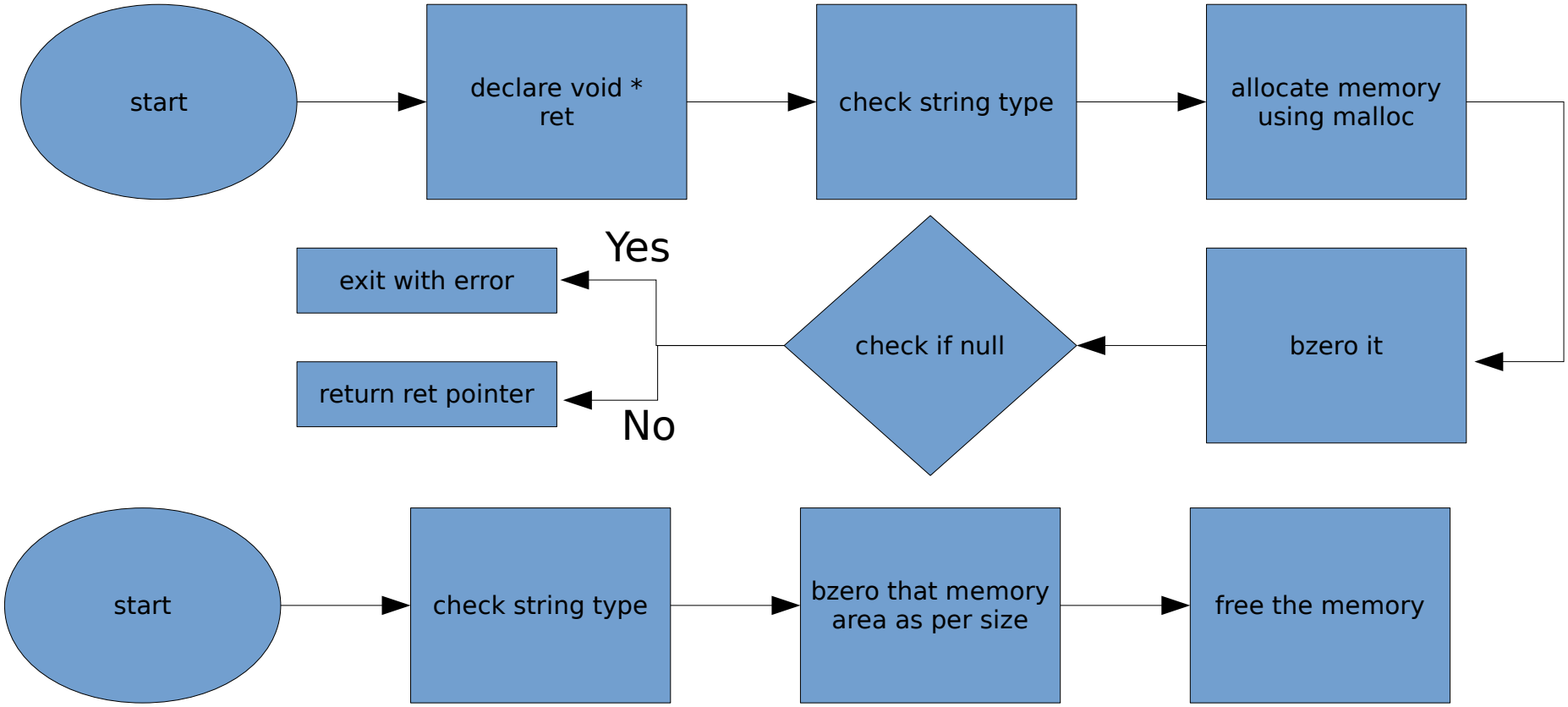
- int snd(int sock, char *cmds, char *reason, char *retval)
- char *rcv(int sock, char *reason, char *retval)





allocate.h

- void *allocate(char *type, int size)
- void deallocate(void *a, char *type, int size)



TODO

- Create a good mininet topology script to handle dynamic connection as well.
- Add dynamic detection of new hosts and automatic update in controller database.
- Add caching module in relay for faster access.
- Add redundancy module in relay for higher rate of availability.
- Add sub-relaying support.

Thank You!

Prepared and Presented by:
Naman Arora
RA1511003010235