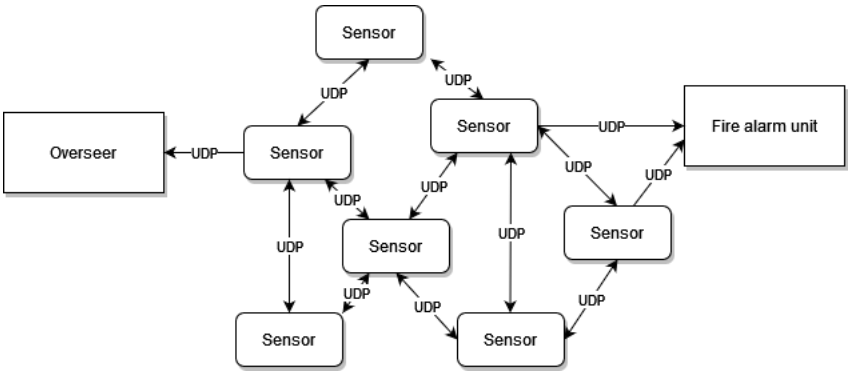
Assessment 2 component: Temperature sensor controller

Summary

Each temperature sensor controller is responsible for a single temperature sensor device, of which there may be many spread throughout the building.

The unique way these are designed is that they form a mesh network, where not all of the temperature sensors are connected to the overseer or the fire alarm unit, but they may be connected to other temperature sensors, and through that can broadcast updates, which will eventually reach those components. This design is simulated through a peer-to-peer network, where updates are sent as UDP datagrams.

Here is a diagram of a possible configuration (although note that many possible configurations exist):



Each sensor controller is launched with a list of IPv4 addresses and ports, and on a temperature change (or after a certain amount of time has passed) will send a UDP datagram containing temperature information to each of the addresses on the list.

Program name

```
tempsensor
```

Command-line arguments

```
{id} {address:port} {max condvar wait (microseconds)} {max update wait (microseconds)} {shared memory path} {shared memory offset} {receiver address:port}
...
```

Shared memory structure

```
struct {
    float temperature;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
};
```

Normal operation

The component will perform the following loop:

1. Lock the mutex
2. Read the temperature
3. Unlock the mutex
4. If the temperature has changed OR if this is the first iteration of this loop OR if the max update wait has passed since this sensor last sent an update:

   A. Construct a UDP datagram (the format is given below) with the sensor's id, the temperature, the current time and with an address list consisting only of this sensor
   B. Send the datagram to each receiver
5. While there are any UDP datagrams waiting to be received
   A. Receive the next UDP datagram and get the temperature, timestamp and address list from it
   B. Construct a new datagram, adding this sensor's address and port to the address list
   C. Send the datagram to each receiver that does not appear on the datagram's address list
6. Lock the mutex
7. Wait on 'cond' (use pthread_cond_timedwait to ensure that the maximum waiting time is the max condvar wait)
8. Loop back to 2

Datagram format

The UDP datagram consists of the 'struct datagram' struct specified here:

```
struct addr_entry {
  struct in_addr sensor_addr;
  in_port_t sensor_port;
};

struct datagram_format {
  char header[4]; // {'T', 'E', 'M', 'P'}
  struct timeval timestamp;
  float temperature;
  uint16_t id;
  uint8_t address_count;
  struct addr_entry address_list[50];
};
```

The timestamp is in the `struct timeval` format used by the gettimeofday() function in <sys/time.h> (type `man gettimeofday` for details) - it is a high resolution timestamp featuring the time in seconds and microseconds.

The address list is used so that sensors do not accidentally propagate a temperature update to a sensor that has already seen it. The sensor originating the temperature update will include its address at the start of the address list, then each sensor receiving the update will add its own address to the list. The 'address_count' field contains the number of addresses in the list. If the address list is full upon a sensor receiving it (address_count is 50), the sensor will remove the first item in the address list (shifting all the other addresses back), then add itself to the end of the address list, keeping address_count at 50. This means the oldest address on the list will be removed, which in most cases should not cause any problems.

Example operation

The program might be executed from the command-line with the following:

```
./tempsensor 801 127.0.0.1:6001 10000 1000000 /shm 1200 127.0.0.1:6002 127.0.0.1:6003 127.0.0.1:6004
```

The program will shm_open shared memory segment at /shm with an offset of 1200 and the size of the struct defined above, and mmap it into memory. It will also bind UDP port 6001.

It will then lock the mutex, retrieve the current temperature, then unlock the mutex.

As this is its first time reading a temperature, it will construct a UDP datagram featuring the current time (retrieved with gettimeofday), the temperature that it read previously, its id number (801), an address_count of 1 and an address_list containing a single address and port: 127.0.0.1 (in struct in_addr format, which is in network byte order) and port 6001 (in network byte order). It will then sendto() this datagram to 127.0.0.1:6003 and 127.0.0.1:6004.

It will then call recvfrom() (passing MSG_DONTWAIT as a flag to ensure the call does not block) to retrieve a waiting datagram, if there is one. If recvfrom() returns a positive value, this program will add its own address and port to the address list of the datagram, then send it out to 127.0.0.1:6003 and 127.0.0.1:6004 (assuming neither of those are already in the address list. Any receivers that are in the address list will be skipped)

The program will lock the mutex again, do a pthread_cond_timedwait on the 'cond' field in shared memory for 10000 microseconds, and then retrieve a new temperature.

As the program has already read in a temperature before, it will check the new temperature, and only send an update if the temperature has changed or if it's been 1000000 microseconds since the last time it sent an update. Otherwise, it will skip straight to calling recvfrom() to check for new packets.