

Message Hub (AI). Draft updated 08/06/02.

Introduction

This document is a quick update of Artem's last draft version. It's not finished, but contains all anticipated section headings and should be useful until the final version appears. I've not massaged the English in all the sections. (CY 17/06/02).

The purpose of this document is to describe how the message Hub program of the MVD online message passing system works.

This document is primarily addressed at those people who need to maintain the Hub and who require a detailed understanding of the internal structures and methods used. It should also be of interest to those users writing Hub client programs. For a better understanding of the format and exchange protocol of messages passed between processes participating in the MVD online message passing system it is recommended that the [MVD message formats and protocols](#) document be read before reading further.

Outlined text highlights programming information likely to be useful for people maintaining the Hub.

The contents are:

- [Hub purposes](#)
- [Hub Structure Overview](#)
- [Where to find the source code](#)
- [Principle data structures](#)
- [Hub threads](#)
 - [Gatekeeper thread](#) {empty}
 - [RWorker thread](#) {empty}
 - [Brain thread](#) {empty}
 - [WWorker thread](#)
 - [Grave thread](#)
- [Mutex details](#)
- [Queue details](#) {empty}
- [Forwarding details](#) {empty}
- [Signal handling](#) {empty}
- [Hub monitoring](#) {empty}
- [Hub testing](#) {empty}
 - [Test environment](#) {empty}
 - [Using the DDD debugger](#) {empty}
 - [Debug printout](#) {empty}
- [Known problems](#)
- [Improvements](#) {empty}
- [Conclusions](#) {empty}

Hub purposes

The Hub is the central connection point of the MVD online message passing system and fulfills the following functionality issues:

- **Public service name** - clients can acquire a public service name, which is guaranteed to be unique, and subsequently have messages forwarded to clients.
- **Hidden service name** - on connection clients are automatically assigned a unique non public name which allows direct message delivery between connected clients.
- **Forwarding** - client message forwarding requests, based on source public service name and message type identifier, are registered and deregistered by the hub.
- **Message storage** - the hub can store messages, of named public services, which are delivered on forwarding registration.

Clients pass messages between each other via their connections to the Hub thus the hub is a single point of failure and must be as robust as possible.

A prototype hub program, called the collector, was written as a testbed for the features of the proposed hub. The collector provided the functionality described above: client connections were handled with blocking network read and write functions, select was used to decide which client had message reads pending and once read messages were handled (stored, forwarded, etc.) without queues. The principle difficulties encountered were:

- uncontrollable accumulation of network send delays when forwarding the same messages to many clients,
- recognizing that a client connection had been silently lost when the later was designed to push messages into the collector, but was not receiving messages,
- some initial confusion and assumptions regarding how the TCP stack code actually worked for features like `SO_KEEPALIVE`, socket closure, etc..
- the importance of being able to perform concurrent read and write operations on sockets, and
- the absurdity of using blocking io outside of a multi-threaded environment where it is ideal.

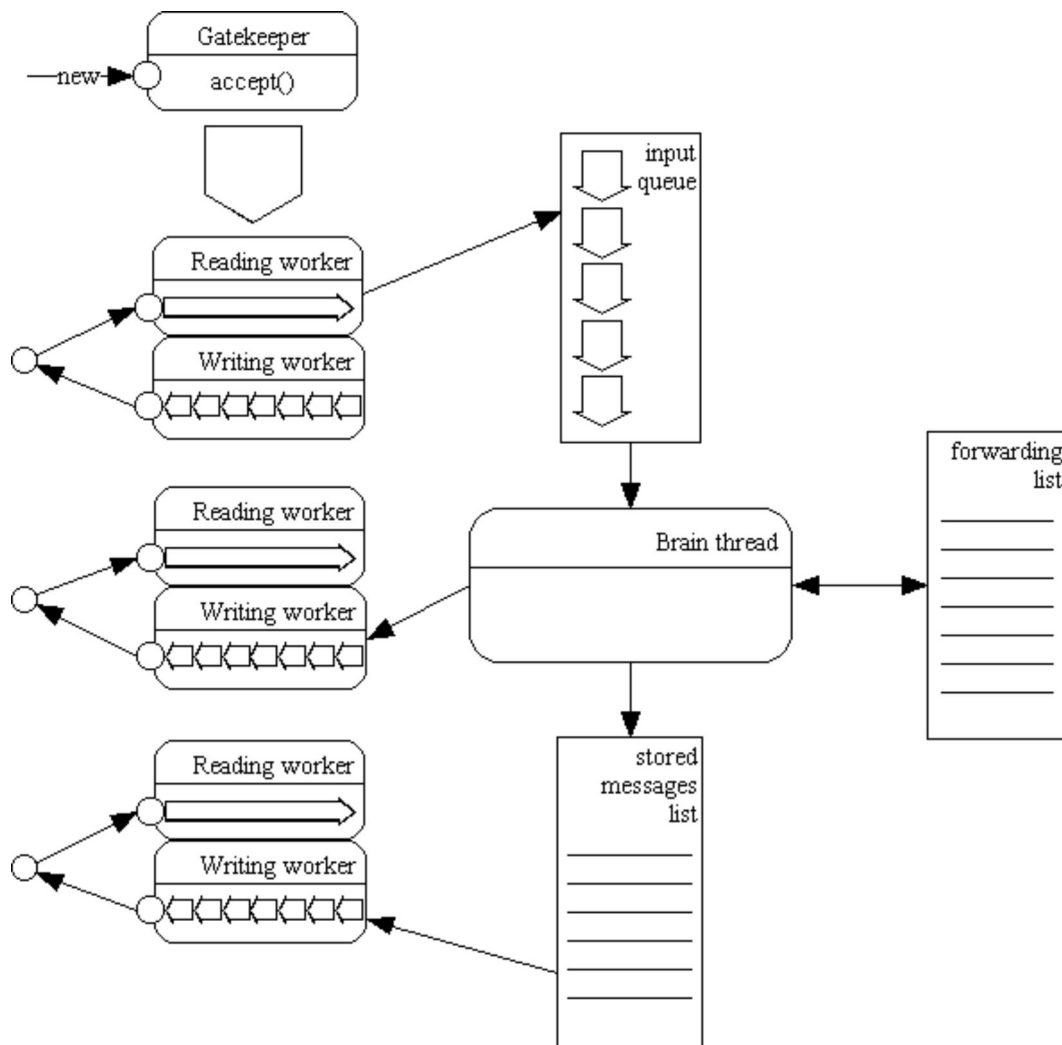
These problems can most elegantly be solved within the multi-threaded programming model. Independent threads can be created for every connection, one reading and one writing, thus providing a safe way of avoiding delay accumulation or blocking problems on concurrent message transmission and reception. A heartbeat thread can be readily introduced to identify silent client disconnections. Additional threads can be created which handle the message forwarding and storage. There also exists the possibility of setting different priorities to threads. These and other points will be discussed in the following sections.

Hub Structure Overview

The hub is multithreaded program executing several command flows (threads) simultaneously. Currently the following threads are used:

- **Gatekeeper** - which waits on the accept socket for new connections.
- **Reading thread** - reads messages from the client.
- **Writing thread** - sends messages to the client.
- **Brain thread** - handles message forwarding and storage.
- **Grave thread** - periodically checks connection activity on every client socket, disconnecting those where the connection has been silently lost.

Thread interactivity is shown in schematically in Fig.1.



Briefly hub operation, excluding error handling, works as follows. New connections are accepted by the Gatekeeper which starts the client RWorker. RWorker starts WWorker which means that two threads are created for every client connection. Messages read by RWorker are decoded and a jacket structure is created where hub relevant information about the message and the decoded and undecoded message are stored. The message jacket structure is inserted in the Brain threads input queue provided it is not a special message (refer to the RWorker description below). The Brain removes message at the head of its input queue and processes them. Most messages are simply forwarded to clients, but some messages may be transferred directly to other connections. In either case the Brain does this by inserting the message into the appropriate WWorker queue. The Brain also determines whether message should be stored after forwarding.

Brain behaviour is modified if the incoming message is a: begin service, forwarding, toservice or end service request (refer to the Brain description below).

The WWorker reads its input queue sending all messages to the client socket.

Asynchronously the Grave thread checks heartbeat timeouts for all client connections closing dead ones.

[Where to find the source code](#)

The hub folder contains the following source and header files:

- **Source.c** - contains utility routines, for creating, reading, sending, decoding and encoding messages, checking name validity, delay routines, etc.
- **Source.h** - header file that contains most declarations. All used structure types are declared here, as are hub tuning parameters, like timeouts, printouts, etc.
- **Hub.c** - contains the main() routine and prepares the principle data structures and starts all the required threads;
- **Grave.c, Grave.h** - Grave code and header files.
- **Gatekeeper.c, Gatekeeper.h** - Gatekeeper code and header files.
- **Brain.c, Brain.h** - Brain thread code and header files.
- **RWorker.c, RWorker.h** - Reading worker code and header files.
- **WWorker.c, WWorker.h** - Writing worker code and header files.

- **Queue.c, Queue.h** - queuing code and header files.
- **Forward.c, Forward.h** - forwarding code and header files.
- **HubStat.c, HubView.c** - hub statistics and viewing client code.
- **HubTest.c, HubTestSource.c, HubTestSource.h** - test programs used for developing and debugging Hub.

Principle data structures

This section describes the principle data structures.

Hub data structures

Main structure in the Hub is HubStruct. One can find its definition in **Source.h** header file. It has only one instance in the Hub and almost in every point can be referred as "Hub". All other structures and data, as any Worker thread, any message, any piece of data can be found starting from HubStruct.

```
struct HubStruct
{
    pthread_t GateKeeper;           // Pointer to gatekeeper thread
    pthread_t Brain;               // Pointer to brain thread
    pthread_t Grave;              // Pointer to grave thread

    ForwardStruct *Forward;        // Queue of forward requests (forward list)
    pthread_mutex_t *ForwardMutex; // Mutex to access forward structure
    char ForwardMutexOwn[100];     // Who locked the mutex

    WorkerStruct *Worker[FD_SETSIZE]; // Workers structure. Note that
                                     // brain is Worker0 and storage is Worker1

    pthread_mutex_t *BrainQueueNumMutex; // Mutex to access BrainQueueNum array
    char BrainQueueNumMutexOwn[100];     // Who locked the mutex
    unsigned short BrainQueueNum[FD_SETSIZE]; // Array that keeps number of messages
                                             // in Brain queue received from different sockets

    pthread_mutex_t *WorkerMutex;      // Mutex to access workers structure
    char WorkerMutexOwn[100];          // Who locked the mutex
    int Workers;                       // Number of currently running workers
    // Brain thread and storage are not included

    int MaxWorker;                    // Maximal socket that is used

    char *ServiceName;               // Service name of hub
    INFO_source *Source;             // Host address and port of hub
};
```

Lets focus on some points in this structure.

```
ForwardStruct *Forward;           // Queue of forward requests (forward list)
pthread_mutex_t *ForwardMutex;    // Mutex to access forward structure
char ForwardMutexOwn[100];       // Who locked the mutex
```

Here one can see pointer to the queue that stores all forwarding requests from all connections. Mutex is used control access to this structure. You can find more information about using mutexes in the Hub in [Mutex details](#) chapter. Forward list is made as a linked list. A more comprehensive description of forwarding can be found in [Forwarding details](#)

```
WorkerStruct *Worker[FD_SETSIZE]; // Workers structure. Note that
                                     // brain is Worker0 and storage is Worker1
```

Array that contains pointers to all worker's structures. Socket in unix

Hub threads

Some notes about possible number of threads.

You can read [LinuxThreads Frequently Asked Questions](#) or [Lambda Computer Science FAQ](#) for better understanding of the question.

There are several different models for building multithreaded server:

- The simplest structure is server which creates one thread for each connection which performs all reading, processing

and writing operations;

- If number of connection is huge (thousands) there could be a fixed-size pool of worker threads that pick incoming connection requests from a queue;
- In our case server creates two different threads for every connection, one performing all reading, another performing all writing operations.

The specific of MVD message parsing system is that there could be at most hundred connections. Some of connected client can exchange rather big messages. Histograms could have size up to several megabytes. When using thread pool there is possibility that most of worker thread will be busy by reading large messages from clients with slow connections, and other requests will be delayed. So it's better to have single thread for every connection, which can spend much time reading the network. Another point is that client could be desined so that it prefers, for example, writing when it should read. To avoid situation when both sides try send message Hub has two different threads for reading and writing single socket. With this inner structure there should be enough computer power to run expected number of threads.

Gatekeeper thread

This section is empty.

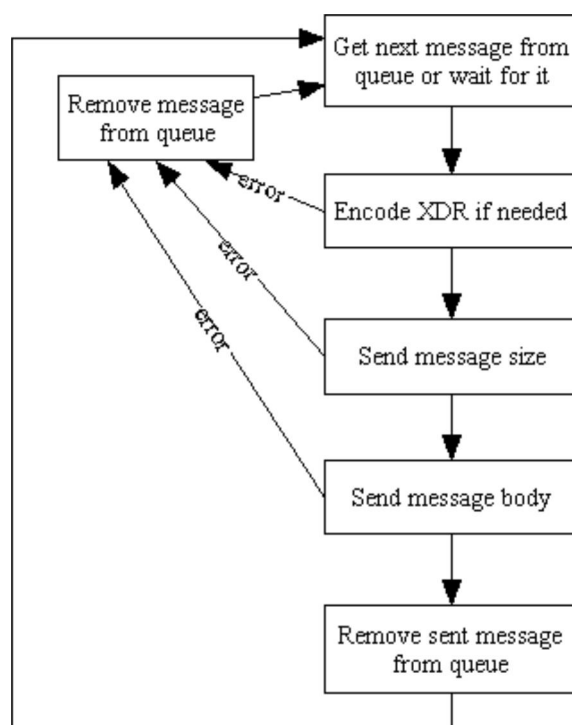
RWorker Thread

This section is empty.

Brain thread

This section is empty.

WWorker Thread



Writing thread (WWorker) is extremely simple. It has a queue of message that should be sent to socket. WWorker runs in infinite loop checking this queue. If queue is empty it just sleeps on `pthread_cond_wait()` call. If Brain or RWorker puts some message to the queue, WWorker wakes up and starts sending this message to socket. After successful sending WWorker checks for the next message and if the queue is empty it continues waiting.

If WWorker is delayed while sending a big message through the slow connection there is still possibility for RWorker to stop it. RWorker closes socket. In this case WWorker returns from `send()` call with error and stops on `pthread_testcancel()` until RWorker will join it.

The code highlights for WWorker are:

```
WWorker(WorkerStruct W)
{
    // Main writing worker cycle
    while (TRUE)
    {
        // This a point where WWorker can safely cancel
        // if RWorker has called pthread_cancel()
        pthread_testcancel();

        // Get first message from the worker queue
        Message = QueueGetFirst();

        // Encode XDR message if needed
        if (EncodeMessage(Message) <= 0)
        {
            QueueRemoveFirst();
            continue;
        }

        // Send message to socket
        if (!SendMessage(W->Socket, Message))
        {
            shutdown(W->Socket);
            while (TRUE) pthread_testcancel();
        }

        // Remove sent message from worker queue
        QueueRemoveFirst);
    }
}
```

Grave thread

There is a certain timeout defined in a system (look **tcp_keepalive_time** value at "man tcp"), after which the connection with no any activity is assumed to be broken. In Linux by default this timeout is set to at least 2 hours. This means that if client has been disconnected it is possible that Hub will never understand this fact for such a long time. For Hub it'll be equal as there is just no any messages going from this client. If this client would have an acquired public service name, which is unique for the whole system, nobody else will be able to use this service or to acquire this service name. So there is a possibility that service can be unusable for more than 2 hours.

Grave thread is used to avoid such long timeouts on network connetions thus providing more reliability to the whole system. For this reason Hub periodically (ALIVETIME defined in Source.h file) sends INFO_HEARTBEAT message. Every client is requered to answer on this message with INFO_ACK_HEARTBEAT message.

To avoid long delays in connections Hub has to have some keepalive mechanism. There is a comprehensive explanation of this question in "[Unix Socket FAQ](#)". Grave thread is implementation of this mechanism.

Every worker structure associated with connection has a field W->Alive were Hub keeps the time when the last message from this connection was receied.

```
struct WorkerStruct
{
    ...
    long Alive;                // Last time when worker was alive
                                // Updated on every received message
}
```

On every received message RWorker updates this value. In RWorker.c you can find something like this:

```
// Receive message
if ((Message = ReceiveMessage(W->Socket, W->Hub)) == 0x0) pthread_exit(0);

...

// Update the value when the last message was received. Used by Grave thread
gettimeofday(&TimeAlive, 0x0);
W->Alive = TimeAlive.tv_sec;
```

Grave thread periodically compares this time value with current time. Source.h file contains definition for the period of these checks.

```
#define ALIVETIME 120
```

At first Grave thread looks if the last message from this connection was received too long time ago. If this time is more than ALIVETIME timeout it closes connection.

If only half of this time gone Grave thread sends a heartbeat message to the client thus showing it that it should send some message to prove that it's still alive. If client will never reply on heartbeat message it'll be disconnected on the next loop cycle.

Theoretically we can update alive time value not only when receiving message, but when sending messages. It is possible that system will return error code on send() call if connection is broken giving us understanding of this fact. But we can't be sure because TCP stack has some buffer and when connection is lost, message that we send can stay in this buffer and we'll never receive error code until this buffer will be filled.

So if client sends some messages Hub never sends INFO_HEARTBEAT message to it. If client is silent (most viewing processes are) Hub forces it to send a reply to heartbeat message.

You can get better understanding of Grave's functioning algorithm observing this code highlights:

```
void *Grave()
{
    Delay = abs(ALIVETIME/2);

    // Main grave thread loop
    while (TRUE)
    {
        // First two sockets are standart input and output.
        // As they are not used their numbers in worker list are given
        // to Brain and message storage.
        Count = 2;

        // Run though all possible connections
        while (Count < FD_SETSIZE)
        {
            // If connection under this address never exists go to next step of the loop
            if (Hub->Worker[Count] == 0x0)
            {
                Count++;
                continue;
            }

            // Get current time
            gettimeofday(Time);

            // If wait time for this worker exceeded close connection
            if (Hub->Worker[Count]->Alive < (Time - ALIVETIME))
            {
                // Close connection
                shutdown(Count);
                Count++;
                continue;
            }

            // If only half of wait time for this worker exceeded
            // we should send message to the worker
            if (Hub->Worker[Count]->Alive <= (Time - Delay))
            {
                CreateMessage(Message);
                Message->Info->a.rest.INFO_DATA_U_t = INFO_REQ_HEARTBEAT;
                QueueAdd(Hub->Worker[Count], Message);
            }
            Count++;
        }
        Sleep(Delay);
    }
}
```

Mutex details

As the Hub has several threads that use same resources, as messages and queues, all operations are protected by using mutexes.

Mutex's policy is not very complicated but is very strict, as any wrong mutex usage can lead to abnormal program termination.

Every message has it's own mutex (Message->Mutex) and one must LOCK it to CHANGE something in a message.

Every worker's structure has mutex and one must LOCK it always to READ or WRITE any field.

Before locking worker mutex one must be sure that this worker exists. If there is no guarantee that worker is alive (Hub->Workers[Count] != NULL) one must LOCK Hub->WorkerMutex and check. Otherwise segmentation fault will occur (NULL->Mutex).

Nobody is allowed to change Hub->Workers[] array without locking Hub->WorkerMutex.

If it is needed to lock several worker mutexes (Worker[]->Mutex) one must lock Hub->WorkerMutex before to avoid dead locks.

Normally mutexes are used by calling pthread_mutex_... functions. The Hub has three cover functions for these calls. All of them are defined in **Source.h** file and are made as inline code to be able to write exact place of calling function in source code:

- MutexLock() covers pthread_mutex_lock();
- MutexUnlock() covers pthread_mutex_unlock();
- MutexWait() covers pthread_cond_wait().

Cover functions are extremely helpfull for debugging. On every operation on mutex cover structure prints out detailed information about this operation. Checking printouts one can find information like this: "Thread ... locked mutex Function call was made from source file ... line number ...". So it makes easier to find deadlocks.

Moreover, every mutex has associated string line called MutexOwn in which cover functions write the same information about mutex operations. This information can be viewed in debugger, allowing on every step to know the owner of mutex and where exactly it was locked.

After finishing mutex debugging cover functions can be easily switched off by removing MUTEXVERBOSE definition from **Source.h** header file.

Printout looks like this: **03:31:59 {1026} Mutex 134810536 locked Queue.c:129 QueueGetFirst()**

- **03:31:59** - Time when the function call was made;
- **{1026}** - Thread ID of calling thread;
- **134810536** - Mutex ID allows to distinguish mutexes;
- **locked** - Operation that was made on mutex. Possible variants are:
 - lock - thread tries to lock mutex and enters the state when it is doing nothing until the current mutex owner will release this mutex and thread will own the mutex;
 - locked - mutex successfully owned by thread;
 - unlock - mutex released;
 - unlock (cond) - mutex is unlocked for waiting conditional variable. This is used when message queue is empty and thread goes to wait for some message to come. When message will come to queue conditional variable will be signaled, thread will lock mutex and get the message. The described operation is made in QueueGetFirst() function.
 - lock (cond) - Conditional variable signalled, which means that some message came to queue, mutex locked and thread wakes up and goes to continue.
- **Queue.c:129 QueueGetFirst()** - shows where in source code this function call was made.

Queue details

This section is empty.

Forwarding details

This section is empty.

Signal handling

This section is empty.

Hub monitoring

This section is empty.

Hub testing

This section is empty.

Test environment

This section is empty.

Using the DDD debugger

This section is empty.

Debug printout

This section is empty.

Known problems

Bug report and fix history are listed [here](#).

Improvements

This section is empty.

Conclusions

This section is empty.