

MVD message formats and protocols (AI,CY). Draft updated 01/05/02.

Introduction

The purpose of this note is to define format and exchange protocol of messages passed between processes participating in the MVD online message passing model.

In this note the model is described from top-to-bottom introducing items in the order which is most likely to be of interest to the first time read. The first section contains a review of model features. The second section specifies the behaviour of the message hub. The following section illustrates, with psuedocode examples based on the client function library, how clients interact with the hub. The current message format is then defined. In the final section the message definition versioning philosophy is described followed by a description of previous message definitions.

Text rendition is used to highlight items which have special significance:

- Italicised text indicates the first occurrence of a definition.
- Bold text highlights a hub implementation requirement.
- Italicised bold text are improvement suggestions which possibly violate the current implementation of the hub.

The contents are:

- [Model](#)
 - [Service identification](#)
 - [Message identification](#)
 - [Tag identifier](#)
 - [Hash identifier](#)
 - [Message forwarding](#)
 - [Stored messages](#)
 - [Sticky messages and dead services](#)
 - [Direct message delivery](#)
- [Hub specification](#)
- [Client-hub protocol](#)
 - [Connecting to the hub](#)
 - [Acquiring a public service name](#)
 - [Releasing a service name](#)
 - [Registering a forwarding request](#)
 - [Deregistering a forwarding request](#)
 - [Peeking into stored messages](#)
 - [Sending a message directly to a specified service name](#)
 - [Heartbeat](#)
 - [Disconnection](#)
- [Message format](#)
 - [INFO_MSG definition](#)
 - [INFO_OBJ definition](#)
 - [INFO_DATA_U definition](#)
- [Versioning](#)
 - [Version 0.0](#)

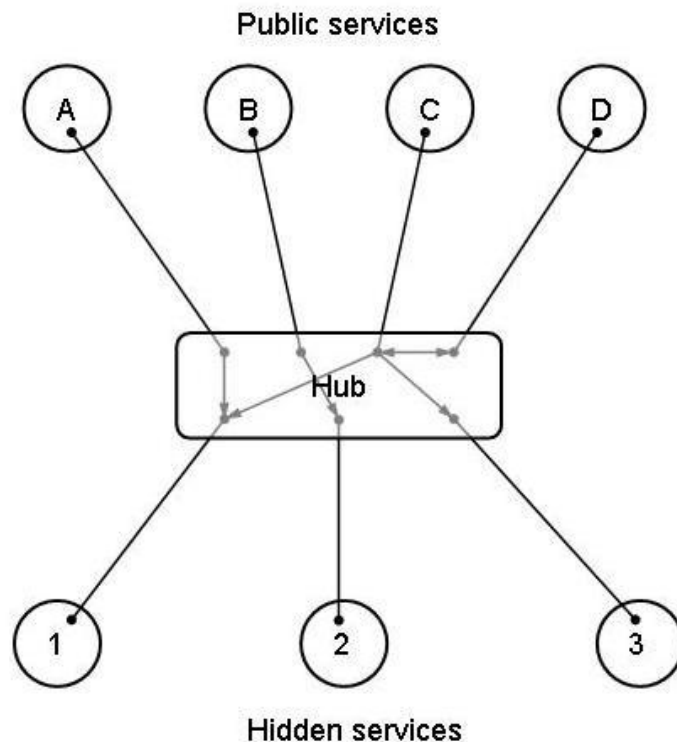
Model

The message passing model used is based on a central *hub* process which is the single connection target for participating *client* processes. This has inherent advantages a single address specifies the hub which is the switching yard of all messages transmitted. It's disadvantages are manifest the hub is a single point of failure and hub loading is likely to be an issue. The communication protocol used is TCP/IP which is connection oriented and ensures that data sent is received in the correct order.

The functionality offered by the model is:

- name service functionality allowing hidden and public named client connections,
- message forwarding based on source public service name and message type identifiers,
- message storage at the hub of messages, and
- direct message delivery between connected clients.

These are illustrated in Fig. 1.



Service identification

A *service name* is used to identify clients to the hub and other clients.

The service name character string must satisfy the following requirements:

- it must contain at least one valid character,
- valid characters are alphanumeric (a-z,A-Z,0-9), hyphens or underscores.
- the maximum number of characters is INFO_MAX_STR (currently 128).

Service names can be either *hidden* or *public*.

A *hidden* client service name is automatically assigned when a connection is made to the hub. The name assigned is the port@host (where port is the port number and the host is the dotted ip address, eg. 12345@127.0.0.1) of the client's socket. Hidden service names are unique and identifiable as they contain the, otherwise, illegal @ character. The service name is said to be hidden because other clients are not informed of the begin or end of a hidden clients connection.

A *public* service name, specified by the client and guaranteed by the hub to be unique, can be *acquired* and *released* by a hidden client as described in the protocol section later. Public service names are used by clients which provide information used by other clients. The original hidden service name is reinstated when a public service name is released.

The service name is appended, by the client side software, to all messages send to the hub and forwarded by the hub.

Message identification

Hub message handling is based on the service name, described above, an enumerator constant tag and hash field identifiers described below. All identifiers are inserted by the sender client into the message.

Tag identifier

The tag identifier is an enumerator constant which determines the message type (payload content). The tag is generated automatically and is not defined by client or hub.

Hash identifier

The hash identifier is a 128 bit field which can be used to hash the message content thus making different message content identifiable even though service name and tag identifiers are identical. This is particularly useful for histogram messages of the same dimensionality (same tag) which need to be stored as separate messages by the hub, see below. By default all bits of the hash field are clear. The choice of a 128 bit field allows the use of popular encrypting tools, eg. md4 and md5, although this is not required.

Message forwarding

The hub forwards a received message to a client when the message identifiers match with one of the clients previously registered forwarding requests. The forwarding syntax used when specifying forwarding request doublets (public service name, tag) are shown in the following table. Clearly names conform to the service name requirements already stated with the exception of the * wildcard.

name	tag	Messages forwarded
abc	0	All messages with name abc irrespective of tag
abc	N	Messages with name abc having tag N
abc*	0	All messages where the first three characters of the name are abc irrespective of tag
abc*	N	Messages where the first three characters of the name are abc having tag N
*	0	All messages
*	N	All messages having tag N

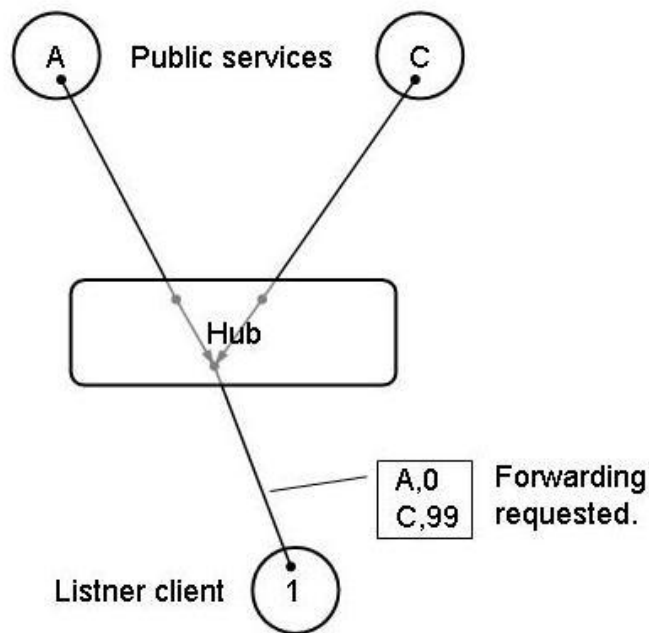
Triplet (public service, tag, hash) forwarding requires specifying the hash field. The requests hash field is, by default clear, which selects all hashes. If it is set then the doublet selection tabulated above is made with the additional requirement of an exact match of the hash field. **Currently hash specific forwarding requests can only be removed by deregistering the associated doublet or all requests.**

A registered forwarding requests can be deregistering by specifying the appropriate name and tag in a deregister request.

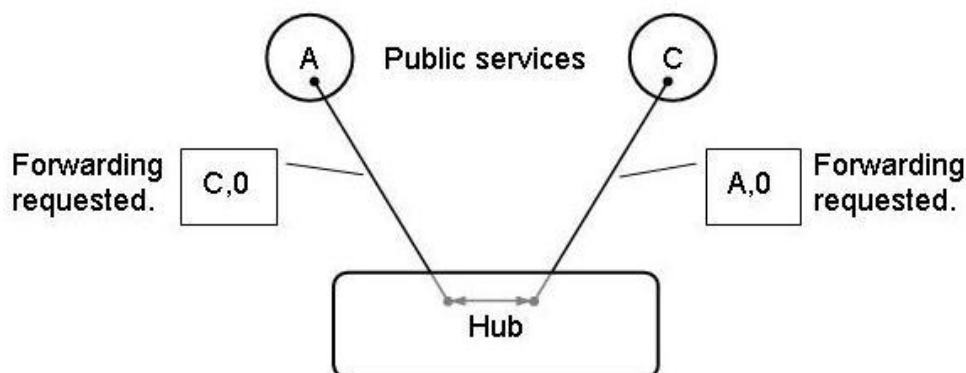
The following steps illustrate a typical clients usage of forwarding:

- on establishing a connection no unsolicited will be received until a forwarding request is registered.
- the required forwarding requests are registered eg. (name=abc,tag=0), (ij*,12345) and (xy,0).
- for each successful request, the hub will reject requests with illegal service name characters, etc., the hub forwards messages that match the request as follows:
 - stored messages at the hub are sent, see next section, followed by,
 - messages which arrive at the hub thereafter.
- the hub ensures that messages are delivered only once even if they satisfy multiple forwarding requests.
- the client may add or remove requests during its lifetime.
- forwarding requests can be removed by specifying the exact (name,tag) pair, eg. (abc,0), (ij*,12345) or (xy,0).
- if (*,0) is deregistered the hub clears all forwarding requests and the knowledge of which messages have been already been delivered to the client from storage, later forwarding requests are treated as if the connection is newly created.

The communication protocol between connected hub clients is defined by the forwarding requests they make. A listening client such as a viewer need only request those messages which he is interested in receiving as shown in Fig. 2.



Two clients which exchange information between each other need only specify the other client service in their forwarding requests as shown in Fig. 3.



The hub client protocol used to *register* and *deregister* forwarding requests is described in the protocol section.

Stored messages

The hub has the ability of storing the latest version of each message, identified by service name, tag and hash, in the order it is received. Stored messages are sent to connections in the following cases:

- If a successful forwarding request is made that matches stored messages in which case these are sent before messages received later. This is useful for viewer services who connect, get the current state and disconnect.
- If an acquire service request is accepted which specifies that previous dead service sticky, see below, messages are to be received. This is used by client processes to remember their last known state when recovering from a crash (assuming the hub has not crashed).

Sticky messages and dead services

Sticky messages are messages which remain in the store after the service name acquired to create them has been released. This feature is useful if information is to remain available, such as configuration settings, or if the next user of the same public service name should receive context information from its previous incarnation. The sticky property is an option of the begin service request.

A *dead service* does not exist, the name refers to the variable set in the message format INFO_OBJ structure which informs a process receiving the message that the service name which created the message is no longer assigned.

Direct message delivery

In some, rare, cases it is necessary for clients to be able to send messages to public services irrespective of the forwarding requests the service has made. This is described in the protocol section.

Hub specification

- Begin service
 - **The hub enforces the valid service name definition and disconnects clients using illegal names**
 - **The hidden service name, port@host, is assigned by both client and hub side software when a connection is established.**
 - **The hub never generates or forwards begin or end service messages for hidden service names.**
 - **The hub enforces exclusive service name usage, that is a service name when acquired is guaranteed to be unique.**
 - **The hub allows only one service name to be associated with a connection.**
 - **The hub and connection side software reassign the original hidden service name when a service name is released.**
 - **The hub allows both hidden and public service names to be the end point address of direct, toservice, messages.**
 - **The client must wait for the acknowledgement of a begin public service request before sending additional messages.**
 - **If the hub detects a service name mismatch it closes the connection. *Currently no error message is associated with this.***
 - **A begin service request made by a connection which has already acquired a public name is treated as an error by the hub and the connection is closed.**
- End service
 - **An end service request made by a hidden connection is treated as an error by the hub and the connection is closed.**
 - **The hub generates and forwards an end service message for public connections if they close without sending the end service message.**
- Forwarding
 - **The hub forwards only public service messages.**
 - **The hub refuses incorrect forwarding requests (eg. illegal characters in the name).**
 - **The hub refuses multiple identical forwarding request.**
 - **A message is forwarded only once to a client.**
 - **The hub sends no unsolicited messages to a connection, if no forwarding requests are registered.**
 - **The hub accepts forwarding request associated with, as yet, unknown services.**
- Stored
 - **The hub stores only public service messages.**
 - **The hub delivers messages in their stored order, that is oldest first and newest last.**
 - **The hub removes stored messages associated with a service name when the name is released, unless they are**

sticky.

- A stored message is forwarded only once to a client.
- The hub implements selection using service name, tag and hash matching.
- Sticky
 - The hub must set the dead service variable when the service name is released for those service messages in the store.
 - The hub must remove, after performing any requested delivery, stored dead service messages when the service name is reassigned.
- Direct message delivery
 - If the hub cannot deliver the message because the service name does not exist an INFO_FAIL_TOSERVICE message is returned to the sender with the original request is piggybacked to it.
- Disconnection
 - If the hub crashes or decides that the connection is blocked the connection is closed.

Client-hub protocol

This section defines all cases where hub and process communication occurs which is not straight forward message delivery.

Connecting to the hub

A connection to the hub is established by connecting to the hub's known address.

```
if (info_init(info_connect(my_port,my_node))) {
    printf(stdout,"hub connection established\n");
} else {
    printf(stdout,"hub connection refused\n");
}
```

Acquiring a public service name

To acquire a service name a connected process sends a begin service request, tag INFO_BEG_SERVICE, specifying the name required to the hub.

```
{
    INFO_MSG msg;

    info_zero_msg (&msg);
    msg.a.rest.INFO_DATA_U_t = INFO_BEG_SERVICE;
    msg.a.rest.INFO_DATA_U_u.begsrv.name = "fred"; // request service name fred
    msg.a.rest.INFO_DATA_U_u.begsrv.sticky = 1; // set the sticky bit
    msg.a.rest.INFO_DATA_U_u.begsrv.receive = 1; // receive dead service messages of

    info_send_msg (&msg);
}
```

The hub acknowledges with a INFO_ACK_BEG_SERVICE message the status of which identifies whether the request has been satisfied or not. *The original request is piggybacked to the acknowledgement.* The client must wait for the acknowledgement before sending additional messages.

```
{
    INFO_MSG msg;

    info_zero_msg (&msg);
    info_read_msg (&msg);

    switch (msg->a.rest.INFO_DATA_U_t) {

    case INFO_ACK_BEG_SERVICE:
        if (!msg.a.status) printf ("begin service request failed\n");
    }
```

```

    printf ("my service name is: %s\n",info_get_service());
    break;

default:
    break;

}
}

```

Releasing a service name

To release a service name the connected client sends an end service request, tag INFO_END_SERVICE, to the hub. For historical reasons the service name is specified in the message.

```

{
    INFO_MSG msg;

    info_zero_msg (&msg);
    msg.a.rest.INFO_DATA_U_t          = INFO_END_SERVICE;
    msg.a.rest.INFO_DATA_U_u.end_srv_name.str = "fred";           // service name to release

    info_send_msg (&msg);

    printf ("my service name is: %s\n",info_get_service());
}

```

No acknowledge message is associated with the release service request.

Registering a forwarding request

Forward service request, tag INFO_REQ_FORWARD, specifying the name and tag identifier pair to the hub.

```

{
    INFO_MSG msg;

    info_zero_msg (&msg);
    req.a.rest.INFO_DATA_U_t          = INFO_REQ_FORWARD;
    req.a.rest.INFO_DATA_U_u.req_frw.str = "abc*";           // forwarding name match required
    req.a.rest.INFO_DATA_U_u.req_frw.nr = 0;                 // forwarding tag match required

    info_send_msg (&msg);
}

```

The hub acknowledges with a INFO_ACK_FORWARD message containing the status of which identifies whether the request has been satisfied or not. ***The original request is piggybacked to the acknowledgement.***

```

{
    INFO_MSG msg;

    info_zero_msg (&msg);
    info_read_msg (&msg);

    switch (msg.a.rest.INFO_DATA_U_t) {

    case INFO_ACK_FORWARD:
        if (!msg.a.status) printf ("forward request failed\n");
        break;

    default:
        break;

    }
}

```

Peeking into stored messages

The INFO_PEEK_FORWARD and INFO_ACK_PEEK_FORWARD request acknowledge pair produce similar results as the

forwarding message above, but do not install permanent forwarding requests only the stored messages are checked for matches.

Deregistering a forwarding request

To deregister a forwarding request the connected process send a remove forwarding request, tag INFO_RMV_FORWARD, to the hub.

No acknowledge message is associated with the deregister forwarding request.

Sending a message directly to a single service name

A message can be sent to a specific service name by assigning the toservice field in the INFO_OBJ structure.

```
{
    INFO_MSG msg;

    info_zero_msg(&msg);
    msg.a.toservice = "fred";           // target for direct delivery
    ...
    tag of message to send and payload contents.
    ...

    info_send_msg (&msg);
}
```

The INFO_FAIL_TOSERVICE mechanism is provided to reduce possible timeouts which might occur between processes which use the toservice delivery mechanism to communicate with each other.

Heartbeat

Both hub and client side software need to be aware of *silent* disconnections resulting from network disruption or host crashes when the in-built TCP end connection mechanism can fail. To avoid silent disconnections the hub sends a heartbeat message, INFO_REQ_HEARTBEAT, to a client when no messages have been received within the previous beat period. The client must reply, within the next beat period, with a INFO_ACK_HEARTBEAT message otherwise the connection is closed.

The connection side software uses the heartbeat to identify lost connections. This is particularly important for listening only connections.

```
for (;;) {
    isel = select (MakeSelectMask(), &rmask, NULL, NULL, &timeout);
    if (!isel) {
        if (!info_network_activity()) printf ("no network activity - silent connection loss ?\n");
    }
}
```

Disconnection

On restoring a connection all client side messages (begin service, state, etc.) must be retransmitted so that the client state is completely recovered.

Message format

Messages are of type INFO_MSG which provides a holder for the total length of the message, the mandatory message object, and, optionally, an array of additional message objects.

Message objects have type INFO_OBJ and contain protocol and payload sensitive data. Protocol information determines who sent the message to who and when. Payload information contains the message content.

The tagged data union has type INFO_DATA_U and contains definitions relating to the content of the message. The INFO_DATA_U_t tag is of particular importance as it defines uniquely the message content.

All messages are defined in a single XDR definition file, info.x, and RPCgen is used to generate the header, info.h, and message encoding and decoding parsing functions, info_xdr.c.

INFO_MSG definition

The INFO_MSG structure is:

```
struct INFO_MSG {
    unsigned int bytes;
    unsigned int vers;
    INFO_OBJ a;
    struct {
        u_int b_len;
        INFO_OBJ *b_val;
    } b;
};
```

where:

- o *bytes* is the total message length in bytes,
- o *vers* is the version number of the message,
- o *a* is the mandatory message object, and
- o *b* is an optional array, *b_val*, of message objects of length *b_len*

INFO_OBJ definition

The INFO_OBJ structure is:

```
struct INFO_OBJ {
    int status;
    INFO_date_time time;
    INFO_source source;
    char *service;
    char *toservice;
    char *symbolic;
    int modifier;
    int dead_service;
    unsigned int hash[4];
    INFO_DATA_U rest;
};
typedef struct INFO_OBJ INFO_OBJ;
```

where:

- o *status* the status value associated with the object. Messages containing replies to requests or updates should set the status value to either 0 (failure) or 1 (success),
- o *time* the date and time of object sending is stored as two unsigned integers (yymmdd and hmmmss),
- o *source* the ip address of the object sender stored as two unsigned integers in network byte order (port and host),
- o *service* the service name associated with the object when sent,
- o *toservice* the service name which is to be sent the object,
- o *symbolic* a user specifyable character string,
- o *modifier* a user specifyable integer,
- o *dead_service* if set the service which provided the object is dead, otherwise it is alive,
- o *hash* a user supplied 128 bit, md5 compatible, hash field associated with the payload content.
- o *rest* a tagged data union.

INFO_DATA_U definition

The INFO_DATA_U structure is a tagged data union containing definitions of all known message payloads:

```
struct INFO_DATA_U {
    INFO_DATA_T INFO_DATA_U_t;
    union {
        INFO_service req_service;
        INFO_str_nr req_frw;
        ..
        .. etc, for a complete listing see info.h
        ..
        INFO_service ack_service;
        INFO_str_nr ack_frw;
    }
};
typedef struct INFO_DATA_U INFO_DATA_U;
```

Versioning

Message definition changes, resulting from addition and removal of fields, is tracked by the *vers* field of the INFO_MSG structure. Version handling is driven by the following considerations:

- the upper 2 bytes of vers are the major and the lower 2 bytes are the minor version numbers.
- the current version number is 1.0 (ie. major.minor).
- vers is included immediately after the message byte count which allowing it's use without full XDR decoding of the message.
- the major number must be incremented if: a field is added to or removed from the message definition, a tag is removed, or when the message content associated with an existing tag is changed.
- the minor number must be incremented if a new tag is added.
- vers applies to all the INFO_OBJ message structures of a message, mixed INFO_OBJ messages are illegal.
- client and hub must exchange messages of the same major version number.
- the hub's linked minor version must always be larger or equal to that associated with a client.
- a client's minor version may be less than the hub's linked version as new tags may not be being forwarded to the client.
- a message decode error always results in the decoder disconnection the connection, whether hub or client.

Maintaining readability of archived xdr message files requires that these be migrated to the current major version. Migration, and the tools available, is described in detail [elsewhere](#).

The sections below describe previous message definitions.

Version 0.0

INFO_MSG definition version 0.0

The INFO_MSG structure is:

```
struct INFO_MSG {
    unsigned int bytes;
    INFO_OBJ a;
    struct {
        u_int b_len;
        INFO_OBJ *b_val;
    } b;
};
```

where:

- *bytes* is the total message length in bytes,
- *a* is the mandatory message object, and
- *b* is an optional array, *b_val*, of message objects of length *b_len*

INFO_OBJ definition version 0.0

The INFO_OBJ structure is:

```
struct INFO_OBJ {
    int status;
    INFO_date_time time;
    INFO_source source;
    char *service;
    char *toservice;
    char *symbolic;
    int modifier;
    int dead_service;
    INFO_DATA_U rest;
};
typedef struct INFO_OBJ INFO_OBJ;
```

where:

- *status* the status value associated with the object. Messages containing replies to requests or updates should set the status value to either 0 (failure) or 1 (success),
- *time* the date and time of object sending is stored as two unsigned integers (yymmdd and hhmmss),
- *source* the ip address of the object sender stored as two unsigned integers in network byte order (port and host),
- *service* the service name associated with the object when sent,
- *toservice* the service name which is to be sent the object,
- *symbolic* a user specifyable character string,
- *modifier* a user specifyable integer,
- *dead_service* if set the service which provided the object is dead, otherwise it is alive, and
- *rest* a tagged data union.

INFO_DATA_U definition version 0.0

The INFO_DATA_U structure is a tagged data union containing definitions of all known message payloads:

```
struct INFO_DATA_U {
    INFO_DATA_T INFO_DATA_U_t;
    union {
        INFO_service req_service;
        INFO_str_nr  req_frwr;
        ..
        .. etc, for a complete listing see info.h
        ..
        INFO_service ack_service;
        INFO_str_nr  ack_frwr;
    }
};
typedef struct INFO_DATA_U INFO_DATA_U;
```