AMI 0.1 protocol specifications

Address Memorizing Interface

Specifications version 4.9
11/11/2002

Changelog and TODO

Table of Contents

Definitions

AMI uses

purposes

AMI is a low-level protocol used for two main purposes:

- Maintain the personal routing table filing peers. Indeed, a peer IP address can change at each connexion. Then, AMI will query others peers to get new peer IP-addresses. For instance, few days ago, you communicated with Bob but its IP address has changed. Through AMI, you will ask above all your AMI network to find his new address - assuming he's connected.

- Locate peers matching specified criteria used by over-protocols. For instance, you are using a AMI over-protocol managing an IRC. You want to find all people wanting to speak about cinema for example. AMI will find such people for you. AMI uses AR requests to find people matching criteria. All criteria are stored under XML format open enough to plug any further protocols.

AMI means 'Address Memorizing Interface' because it is used to maintain informations on a peer community. It makes easier to communicate over the Net.

over-protocols

AMI has been thought to be used with some over-protocols which will plug on it. These protocols are AMI-compliant only if they are based on XML and respect some minimal AMI rules. They can be written in any language and not only Java. Communication between AMI engine and protocols are done by AIS : Ami Internal Services (see below). External programs should stay in $AMIPATH/ext directory.

packets types

Three types of packets are exchanged: Address Request (AR) , Address Information (IAI and EAI) and AMIPING.

XML format

All information packets under AMI are represented under XML format to assure maximum performances, simplicity, compatibility of the protocol and facilitate the development of over-protocols in any language. For some performances and architecture reasons, the XML will need to be well formed but not always validated. AMI uses SAX-type XML parser to process large amounts of data. For each AMI packet, corresponding DTD is given in these specifications for information but validation can be done internally for some types of packets.

Port used

AMI protocol is designed to be run on any network port. For example, a user running AMI on port 80 can communicate with a peer on port 8888. Default default port is **1139** (a=1, m=13, i=9) which is >1024 to deal with security restrictions on UNIX (AMI can be launched without being 'root').

Security / privacy features

Note AMI doesn't process or encode packets (except local parameters serialized on local disk). Every AMI packet is sent in clear (actually, packets can be compressed, so not directly readable) because AMI protocol itself doesn't deal with significant information. Most of packets are AMI-internal and just transfer some IP addresses and the only user-specific information are criteria in AR used to broadcast people. 'Real' privacy could be useful at an application level by over-ami protocols.

However, every AMI packet has to be signed to be authenticated at reception. The attribute 'sign' of the tag ami packet is the MD5 hashcode of the query encoded with RSA private key of transmitter. Exception: the AMIping query doesn't need to be authenticated at transmission but must be authenticated at reply. In all specifications below, we assume packets are authenticated before being processed by AMI engine. Otherwise, they are dropped.

AMI requires a passphrase to start. Main information stored on disk (like peer private key) are encoded symmetrically with this passphrase (using blowfish). AMI provides also some services used by over-protocols to use AMI functions. Most of these services requires the AMI passphrase to be executed. Protocol plugged over AMI don't have access to RSA key pair -which is use internally during network packets authentication- but only to the AMI passphrase.

To finish, AMI engine contents security features to protect these protocols. You can find a SSL-like PKI stream (based on RSA 1024 + blowfish) and a symmetrical encoding stream (based on blowfish).

Checksums, signatures and IDs

- Checksums (cs) used in AMI protocols are based on MD5 hashing algorithm. For low-security level parts (AIS authentication), we use a trivial hashing algorithm: **s[0]*63^(n-1) + s[1]*62^(n-2) + ... + s[n-1]** with n, string length and s[i], byte value of i^{th} character.
- Signatures (*sign* tags) are the encoded value of a MD5 checksum using the RSA private key.
- IDs are random values chosen in interval [0 ; 1E+09] and wrote in radix 16.

The routing table (RT)

Each AMI node contains a RT filing others peers ip addresses. This way, every peer has an image of a part of AMI network at a moment. This information is dynamically shared between peers to find new addresses of peers. The RT is stored under XML format on the peer disk.

Address information

This packet propagates through the network the peers specific information. We find two AI types :

- ***External Address Information (EAI)*** : the information comes from an external source. This packet is required at the first connection. Note that EAI is not a standard AMI query but are an AMI service (see AMI Internal Services protocol).
- ***Internal Address information (IAI or AI)*** : the information is AMI-internal without user intervention.

Address Request

This packet propagates through the network the search for a specific peer IP address. This packet is transmitted on peer demand or is routed in others cases.

AMIPING

The AMIPING process establishes the status of an AMI peer. It is based on the transmission and reception of a specific packet which determines the alive status of the connection. To keep the integrity and performances of the protocol, no information about the transmitter is included in these packets.

AMIPING specifications

The amiping packet is used to check connection with a peer. It can't be mixed with any other packets for integrity and performances reasons.

Packet Format

Variables:

Name	Meaning

pu	Replying peer public key (used to authenticate peer)

Transmission format:

```
<amipacket ami_version='0.1' reply_port='port' >
 <amiping />
</amipacket>
```

Reply format:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <amiping pu='value'/>
</amipacket>
```

DTD (validated):

```
<!ELEMENT amipacket (amiping | (iai|ar)+) >
<!ATTLIST ami_packet
   ami_version CDATA #REQUIRED
   reply_port CDATA #REQUIRED
   sign CDATA #IMPLIED >
<!ELEMENT amiping EMPTY >
<!ATTLIST amiping pu CDATA #IMPLIED >
```

AMIping buffer

address
address 1
address 2 ...

We store every amiping query we receive in a pre-buffer in order to drop multiple queries from a same peer (to fight against Denial of Service and for performances). We store the asking peer IP address in amiping_buffer with unicity on IP. AMI engine will read this buffer from time to time to process AMIping replies. The amiping_buffer is processed as a FIFO.
amiping_buffer owns two methods:

- boolean add(*ip address*) to add the ip address in amiping_buffer.
- boolean remove(*ip address*) to drop entry corresponding with *ip*.

Transmission

Variables*:*
ip : AMIPING packet destination IP address.

```
begin
 send(new amiping,ip)
```

```
end
```

Reception

Variables:
ip : IP of the peer who transmitted the amiping query.

```
begin
 if ip not in amiping_buffer then
  amiping_buffer.add(ip)
 end if
end
```

Reply

AMIping replying is done by specific threads which read the amiping_buffer from times to times. If there is any entry in amping_buffer, we reply to first element.

```
begin
 send(new amiping,ip) with
  pu = <own PU>
  ip = amiping_buffer(0)
 amiping_buffer.remove(ip)
end
```

Pinging a peer - *pu amiping(ip)*

This function represents fact of sending an amiping query at a given IP address and to get a public key if the IP address is used by an AMI peer.
Variables:
ip : Destination IP address

```
begin
 send (new amiping,ip)
 while ( transmission time < AMIPING_TIMEOUT and no amiping reply) do
  wait
 end while
 if transmission time > AMIPING_TIMEOUT
  return -1
 end if
 pu=receive(amiping)
 return pu
end
```

We send an AMIping query and we return pu returned (if there is a reply) and -1 if we get a timeout. Note that in AMI engine, there is no real waiting : sending on a side and receiving/processing result on the other side are done by different software modules.

Status checking - *boolean check_peer(pu)*

This is a generic function used by virtually every AMI packet to check if a known peer is still connected. If a peer is still trusted by RT, we just return true. If not, we send an AMIPING query to the peer and we return true if reply is positive. The *MAX_CHECK_INTERVAL* value is alterable. It represents the maximum time a peer is not checked by AMIPING request before sending a query. AMIPING_TIMEOUT is also alterable and represents maximum time took to reach a peer. See amiparameter to get the full list of AMI parameters.
Variables:
pu : public key of the peer we are checking
date : current date

```
begin
 if date - rt(pu).dlc > MAX_CHECK_INTERVAL or rt(pu).ip_status = 0 do
  if amiping(rt(pu).ip) = pu then
   rt(pu).ip_status <- 1
   rt(pu).dlc <- date
   return true      //OK, peer IP address is still valid
  else if amiping(rt(pu).ip) = -1 or amiping(rt(pu).ip) <> pu then
   rt(pu).ip_status <- 0
   return false     //peer is gone away or IP address is now used by another peer
  end if
 else
  return true      //we return true without check
 end if
end
```

We want to check if the peer associated with the public key pu is still connected. If this user has already been checked recently, we trust the routing table without check. If it is a long time since last check, we send an AMIPING packet to IP address corresponding with pu in the RT. Then, we have three possibilities: first, the peer replies and return a public key being equals to pu, the peer passed the checking ; second, the peer replies but returns another pu: it means that old user is disconnected and another peer took this ip address, then the checked peer ip address is no longer valid, entry status in RT is set to '0' ; third, we have no reply in required amount of time, the user must have left and status in RT is also set to '0' but we keep the old ip address because he may reconnect later with the same address.

The Routing Table (RT)

Each peer updates its routing table at the reception of an Address Information packet. The properties describing a peer are : *pu, al, ip, pf, dlc* and *cm*. Primary key of this 'table' is pu : a peer is always identified by its public key.

Notations

- The notation *rt(pu)* represents the peer associated with *pu* public key.
- The notation *rt(*)* represents all peers in the RT.
- The notation *rt(pu) = null* means that there is no entry in RT matching *pu* public key and *rt(pu) <> null* means there is.
- The notation *rt(pu).param* represents *param* value associated with the peer owning *pu* public key (rt(pu).dlc for example).
- The notation *ip_ip, ip_port* or *ip_status* represents a part of the string *ip.* (example: ip_status=0).

Format

Name	*** pu**	**al**	**ip**	**pf**	**dlc**	**cm**
Goal	Public key	Peer alias	IP address	Perf : capability of right reply	Date last check: Date when we did the last ip check	Optional comments ...
Type	String	String (16)	String	Integer	Integer (in seconds)	String (256)

Notes:

- The IP Address will have the following format: '**<IP>:<port>:<status>**'. Example: '*143.23.62.43:1139:0*'

- The public key uniquely describes a peer. However, two peers or more can have the same alias or IP address.

Valid IP status :

Status	Meaning
0	User is no more connected with *ip* ip address or ip address is no more corresponding with specified user but is now used by another one.
1	User is currently connected with *IP* ip address.

The RT is stored under XML format on peer disk. AMI engine is able to import an external RT.

RT Format:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<rt ami_version='0.1' xmlns='http://www.ami.org/rt'>
 <peer pu='value'
       al='value'
       ip='value'
       pf='value'
       dlc='value'
       cm='value'
 </peer>
</rt>
```

DTD (validated):

```
<!ELEMENT rt (peer)* >
<!ATTLIST rt xmlns CDATA #REQUIRED>
<!ATTLIST rt ami_version CDATA #REQUIRED>
<!ELEMENT peer EMPTY>
<!ATTLIST peer
   pu CDATA #REQUIRED
   al CDATA #REQUIRED
   ip CDATA #REQUIRED
   pf CDATA #REQUIRED
   dlc CDATA #REQUIRED
   cm CDATA #IMPLIED >
```

Adding a new entry in RT - *boolean add_peer(pu,ip,al)*

Function *add_peer* allows to add new entries in the RT. Every entry is checked before being added.

Variables*:*
pu : Public key of the peer we are adding
ip : IP address of the peer we are adding
al : Alias of the peer we are adding
date : current date

```
begin
 if amiping(ip)=pu then
  create new entry in RT with:
   _pu = pu
   _ip = ip
```

```
   _al = al
   _pf = 0
   _dlc = date
   _cm = '')
  return true
 end if
 return false
end
```
To add a new entry in the routing table, we first check its validity by sending an AMIPING query to the peer at ip IP address. If this peer replies an AMIPING packet with a public key equals to waited pu, we add the entry and we return true. Else (time out or wrong reply), we return false.

Altering a peer properties - *boolean alter_peer(pu,ip)*

Function *alter_peer* permits to alter - after checking - properties of a peer in the RT.
Variables*:*
pu : Public key of the peer we are altering
ip : Proposed new IP address for the peer
date : current date

```
begin
 if amiping(ip)=pu then
  rt(pu).ip <- ip
  rt(pu).dlc <- date
  return true
 end if
 return false
end
```
An IAI gave us some information about a peer different from what is in our RT. We alter IP address of the peer after sending of an amiping query.

Black-listing

In order to avoid receiving cracked packets or to ignore someone, AMIPING can black-list a peer. In this case, its comment in RT will be (reserved) : *BLACK-LISTED.* When a peer is black-listed, every packet coming from it is simply dropped and we no more send any packet to it. The peer is identified by its public key. The peer can be black-listed in two cases:

- At user request using the *EAI* service (see services below). Just set comment to reserved word *BLACK-LISTED*. Set anything else to stop black-listing of the peer.
- Automatically, when the peer reach a too bad negative performance. By default, this value is -100 but is alterable.

The Result Table

Result table contains information about current searches and store results. It is something internal and temporary, so it don't use an XML format but is simply a java array.

Format:

arid	max	retrieved	results
value 1	[-1]	[3]	pu2,pu2,pu3...
value 2	[10]	[0]	-
value 3	[5]	[0]	-

Variables:

Name	Meaning
arid	AR id
max	Maximum number of results for this AR. When reached, the entry in result_table is dropped. -1 means infinite.
retrieved	Number of correct replies we retrieved.
results	Results themselves : public key values. Additional information (ip) is found in RT for a public key value.

Notations:

- *result_table(arid)* represents the entry in *result_table* associated with *arid*.
- We can access to *max* parameter for example with notation: *result_table(an arid).max*
- Notation *result_table(an arid) = null* means there is no entry in result table matching the *arid* and *result_table(an arid) <> null* there is.

We declare below a list of functions relative to result table.

Declare an AR search - boolean add_search(arid,max)

Function add_search is called to create an entry in the *result_table*.

Variables:
in: arid : ID of the new AR
in: max: maximum number of replies : integer

```
begin
 if result_table(arid) = null then
  add a row with:
   _arid = arid
   _max = max
   _retrieved = 0
   _results = void
  return true
 end if
 return false
end
```

When we send an AR, we create a new 'AR ID' entry with a void 'Results' set.

Add a result in reply to an AR - *boolean add_result(arid,pu)*

Function add_result is called to add an item in 'Results' row of the result_table.

Variables:
arid : ID for the new AR
pu : matching pu

```
begin
 if result_table(arid) <> null then
  if pu not in result_table(arid).results and result_table(arid).retrieved <
result_table(arid).max then
   add pu in result_table(arid).results
   result_table(arid).retrieved <- result_table(arid).retrieved + 1
   return true
  else
   return false
  end if
 else
  return false
 end if
end
```

Add a new item matching AR identified by arid if an arid entry exists and if pu is not already stored inside.

Drop an entry in result table - *boolean del_result(arid)*

Function *del_result* is called to remove an entry in result table. It may be called because we reached max numbers of replies we were expecting or because we decided to stop search.

Variables*:*
arid : ID of AR entry we want to drop

```
begin
 if result_table(arid) <> null then
  remove entry
  return true
 else
  return false
 end if
end
```

We drop the entry in result table if it exists and return true. Otherwise, we return false.

Internal additional functions

This functions allows to:

- Log a query checksum or ID to detect looping packets : *boolean query_log(value)*
- Check if a query checksum or ID is in log : *boolean is_log(value)*

Log a query ID to detect looping packets - *boolean query_log(value)*

Function query_log is used to log IAI or AR in a spool named *query_history* in order to avoid looping packets: if we receive a packet we already processed, we just drop it.

Variables*:*
value : IAI checksum or AR ID

```
begin
 if query_history(value) = null then
  add value in query_history
```

```
  return true
 end if
 return false
end
```

If the history doesn't contain an entry with value, we add one and we return true. Otherwise, we return false.

Check if a query ID is in log - *boolean is_log(value)*

The function is_log is used to check if a query ID or CS is already in log or not.

Variables*:*
value : IAI checksum IAI or AR ID

```
begin
 if query_history(value) = null then
  return false
 else
  return true
 end if
end
```

If the history doesn't contain an entry with value, we return false, otherwise, we return true.

Internal Address Information

This packet is composed of following information: *putr,altr,criteria, ip, arid, iaics.* This information is sent in two cases:

- type 1: **At each new session:** an internal address information is sent to each amipingable peer in order to declare self new IP address. In this case, criteria is own public key and associated protocol is *ami.*
- type 2: **To reply to an address request:** Main information of this packet is IP address of the peer matching AR.

IAI packet format

Variables:

Name	Meaning
putr	Transmitter public key. The transmitter can be peer itself or another peer. If type 1, putr=pu
altr	Transmitter proposed alias, is used at user creation.
criteria	This criteria is protocol-independent and describes criteria a peer has to match to be retrieved. Most of the time, protocol used is 'ami' and the criteria is the public key of the peer whose we return the ip address (address resolution mode).
al	Alias for the peer described by the IAI

ip	IP address for the peer described by the IAI. Format: *IP:Port*
arid	AR ID. If this IAI is used to reply to an AR, arid=ID of the corresponding AR. Otherwise, arid=0
iaics	IAI checksum, used to drop lost or looping packets. One iaics identifies only information {arid,criteria,ip,al} but not information {putr,altr}. We can find the same iaics for two IAI with different transmitters but with the same {arid,criteria,ip,al}.

General packet format:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <iai putr='value'
      altr='value'
      al='value'
      ip='value'
      arid='value'
      iaics='value' >
  <criteria protocol='protocol name'>
   <param name='parameter name' value='parameter value'/>
  </criteria>
 </iai>
</amipacket>
```

If the IAI is type 1:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <iai putr='own pu'
      altr='own alias'
      al='own alias'
      ip='own ip'
      arid='0'
      iaics='new checksum' >
   <criteria protocol='ami'>
   <param name='pu' value='own pu value'/>
  </criteria>
 </iai>
</amipacket>
```

arid value is zero and packet contains only one criteria with protocol 'ami' and parameter 'pu'. *putr* and *altr* are own pu/al in this case. *iaics* is a new checksum for this query.

If the IAI is type 2 to reply to an AR in Address Resolution mode:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <iai putr='own pu'
      altr='own alias'
      al='found peer alias'
      ip='found peer ip'
      arid='associated AR arid'
      iaics='new checksum' >
  <criteria protocol='ami'>
   <param name='pu' value='pu value'/>
  </criteria>
 </iai>
</amipacket>
```

arid value is corresponding AR *arid* value and packet contains only one criteria : protocol 'ami', parameter 'pu'. *putr* and *altr* corresponds to values found in RT to reply to the AR. *iaics* is a new checksum for this query.

If the IAI is type 2 to reply to an AR not in Address Resolution mode:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <iai putr='own pu'
      altr='own alias'
      al='own alias'
      ip='own ip'
      arid='associated AR arid'
      iaics='new checksum' >
  <criteria protocol='amirc'>
   <param name='interest' value='cinema' />
  </criteria>
 </iai>
</amipacket>
```

- *arid* value is corresponding AR *arid* value and packet don't contain any 'ami' protocol criteria but any others types. More information about criteria is available in chapter User criteria *putr* and *altr* are own values because we answer ourselves that we match the AR. *iaics* is a new checksum for this query.

- Note that if we describe ourself in type 2 IAI, we don't give any IP because it will be gotten by socket properties.

DTD (validated):

```
<!ELEMENT amipacket (amiping | (iai|ar)+) >
<!ATTLIST ami_packet
   ami_version CDATA #REQUIRED
   reply_port CDATA #REQUIRED
   sign CDATA #REQUIRED>
<!ELEMENT iai (criteria+) >
<!ATTLIST iai
   putr CDATA #REQUIRED
   altr CDATA #REQUIRED
   al CDATA #REQUIRED
   ip CDATA #REQUIRED
   arid CDATA #REQUIRED
   iaics CDATA #REQUIRED >
<!ELEMENT criteria (param+) >
<!ATTLIST criteria protocol CDATA #REQUIRED >
<!ELEMENT param EMPTY>
<!ATTLIST param
   name CDATA #REQUIRED
   value CDATA #REQUIRED >
```

Criteria rules:

- MAY contain zero or one Address Resolution criteria with protocol ami and parameter pu. If an IAI packet contains one *ami* criteria, it can't contain any other criteria.

- MUST contain at least one criteria.

Note that you don't find mandatory notion in IAI criteria because IAI packets are basically replies to AR and contain only matching criteria from replying peer, so all criteria in IAI are true for him.

Summary:

The following table sums up the three different IAI types we can receive or transmit.

IAI	**IAI replying to an AR** **arid <> 0**	**Address resolution mode criteria**
		Others criteria protocols
	IAI not replying to an AR **arid = 0**	**Address resolution mode criteria**

Analyze an incoming IAI : ***boolean process_reply(iai)***

This is used by the IAI reception algorithm to check if it contains some fresh information. This information can either be:

- IAI with no associated an AR (type 1 IAI) but containing a new set of {pu,al,ip} we don't have in our RT.
- IAI with no associated AR (type 1 IAI) but containing a set of {pu,al;ip} different from what we have in our RT.
- IAI replying to an AR we sent. In this case, this IAI can be reply to an Address Resolution mode AR (with one unique criteria with 'ami' protocol) or reply to an standard mode AR (with one or several criteria on others protocols than 'ami').

Function returns:

Code	Meaning
-1	Information in IAI is wrong
0	Information in IAI is right but not new
1	Information in IAI is right and new

Variables:
iai : incoming IAI
pu: intermediate public key

```
begin
 if iai.criteria_protocol = 'ami' then     //if it is an address resolution mode IAI
  pu <- iai.criteria_pu
  ip <- iai.ip
 else
  pu <- iai.putr
  ip <- IP got by socket properties
 end if
 if rt(pu) = null then     //if we don't know the IAI transmitter
  if add_peer(pu,ip,iai.al) = true then     //if adding in RT is OK
   if result_table(iai.arid) <> null then
    add_result(iai.arid,pu)
   end if
```

```
    return 1     //right and new information
   else
    return -1
   end if
 else if rt(pu) <> null and rt(pu).ip = ip then     //if we know the peer described in IAI criteria and ip in IAI is the same than in RT
   if check_peer(pu) = true then
    if result_table(iai.arid) <> null then
     add_result(iai.arid,pu)
     return 1
    end if
    return 0     //right but old information
    end if
   else
    return -1
   end if
 else if rt(pu) <> null and rt(pu).ip <> ip then     //we know the peer with ip in IAI other than in RT
   if alter_peer(pu,ip) = true then
    if result_table(iai.arid) <> null then
     add_result(iai.arid,pu)
    end if
    return 1
   else
    return -1
   end if
 end if
end
```

We just received an IAI and we have to analyze it to check if this one contains some right and useful information, some right but useless information or some wrong information. Remember we can receive IAI coming from AMI internal engine (address resolution mode, see Internal Address Information) and we can receive IAI as reply to application level queries transmitted via an AR. These type of IAI (type 2) has always arid value different from zero.

if IAI is in address resolution mode (one unique criteria with protocol='ami' and param='pu'), public key associated with ip tag in IAI packet is criteria_pu (putr can be the address of another peer which send you this information). However, if IAI is not in address resolution mode (any other criteria protocol than 'ami'), public key associated with ip tag is putr because each peer matching criteria replies with its own address.

Firstly, we check if IAI describes a peer we don't know at all (no entry in RT). If so, we try to add this new information in RT and if operation succeeds, we add a result in result_table if this IAI is the reply for a known AR.

If IAI describes a peer we know with the same IP address : we check validly of this information and if it is a reply for a known AR, we log it in result_table. Then, we return 0 because it is a right but old information.

Third case : if peer is known in RT but with a different IP address, we check if IAI transmitter is right. If transmitter is right, we alter our RT. If IAI is an AR reply for an AR we transmitted, we fill also AR result table. Then, we return true and new information status. If test returns false, we return wrong information value.

Transmission at the beginning of session (type 1)

Variables*:*
iai : intermediate iai

```
begin
 iai <- new IAI with
  putr = <own pu>
  altr = <own alias>
```

```
  criteria = <own pu>   (1)
  al = <own alias>
  ip = <own ip>
  arid = 0
  iaics = <new cs>
 for peer in {rt(*)} do
  if check_peer(peer.pu) = true then
   send( iai at rt(peer.pu).ip )
  end if
 end for
 query_log(iaics)
end
```

(1): in this case, total criteria is:

```
<criteria protocol='ami'>
 <param name='pu' value='public key' />
</criteria>
```

When we start AMI process, we send the IAI to every connected peer we find in our RT, in order to 'declare' ourselves to others.

Transmission as reply to an address request (type 2)

Here, we send an IAI for AR reply. Note that information of searched peer is not checked here because this method is called only if this information is already right.

Variables*:*
peer: the peer to which we want to reply (AR transmitter or last AR forwarder).
ar : Address Request to which we want to reply.

```
begin
 if check_peer(peer.pu) = true then
  send(new iai at peer.ip) with
   putr = <own PU>
   altr = <own alias>
   criteria = <matching criteria among corresponding AR criteria>   (1)
   al = <found peer alias>
   ip = <found peer IP>
   arid = ar.arid
   iaics = <new CS>
  query_log(iaics)
 end if
end
```

(1) Criteria must contain at least all AR mandatory criteria and a part of AR optional criteria.(Note that an AR must contain at least one mandatory criteria, see Address request (AR) specifications.

We received an address request for a user we know. After verification of destination peer, we send a new IAI with putr=our public key as transmitter public key, altr=our own alias information, criteria=found peer matching criteria, al=found peer alias, ip=found peer IP address, arid=ar ID and iaics= a new checksum. This checksum will be logged to trace looping packet.

Reception

Variables:
iai : IAI we received

pu : intermediate public key
newiai : intermediate IAI

```
begin
 if iai.criteria_protocol = 'ami' then     //if it is an address resolution mode IAI
  pu <- iai.criteria_pu
 else
  pu <- iai.putr
 end if
 if rt(iai.putr) = null then     //transmitter analyze
  add_peer(iai.putr,iai.iptr,iai.altr)   (1)
 else if rt(iai.putr) <> null and rt(putr).ip <> iai.iptr then
  alter_peer(iai.putr,iai.iptr)
 end if
 if is_log(iai.iaics) = false then     //starting IAI analyze
  query_log(iai)
  if process_reply(iai) = 1 then
   rt(iai.putr).pf <- rt(iai.putr).pf +2
   if iai criteria protocol = 'ami' then
     newiai <- new iai with
       putr = <own PU>
       altr = <own alias>
       criteria = <criteria protocol='ami'><param name='pu' value='pu'
/></criteria>
       al = iai.al
       ip = iai.ip
       arid = 0
       iaics = new cs
     for peer in {rt(*) - rt(iai.putr) - rt(pu)} do
      send(newiai,peer)   (2)
     end for
     query_log(newiai)
   end if
  else if process_reply(iai) = -1 then
   rt(iai.putr).pf <- rt(iai.putr).pf - 1
  end if
 end if
end
```

(1) iptr is got by socket properties
(2) to summarize, if we get an IAI with a pu we don't know or with an IP address different from what is stored in RT, we forward the information to every peer we know. We do that to maintain internally the RT and others criteria are not information-pertinent for others peers at this level.
When we receive an IAI, we first analyze information about last sender. If we don't know him, we store - after checking - his information in my RT. Then, we check that this IAI is not looping and that it's the first time we got it. If not, packet is dropped. Then, we check that information about the peer is correct. If not, we decrement my transmitter performance evaluation (the reply could be wrong if searched user disconnected himself since result sending or if this packet have been send by a cracking program trying to affect AMI performances). In the opposite case, we increase the transmitter performance evaluation by two. We store this result in my RT and we send it to every peer we know and which is pingable. Note that we forward the IAI to others people only in the case it is a new peer or a peer whose IP address has changed.

Address request (AR) specifications

This kind of packet is used to request a list of peers matching some specified criteria by sending a broadcast over AMI network. The corresponding reply would be an IAI. Often, this request will be used internally by AMI to find the IP address of a peer we can't contact any more (Address Resolution mode). In this case, the criteria will be the public key of the peer using the IP address we are looking for. In some others cases, the criteria (one or more) will be application-specific. It can be 'Every peer running AMI protocol AMIrc and wanting to speak about cinema (criteria

name: <TOPIC>)' for example.
All criteria (including own public key) are stored under XML format in a file named 'user.xml'. This file is described in this document at User criteria.
Note that AR query runs two different 'philosophies' following the type of search (Address Resolution mode or others protocols criteria). If we use AR in Address Resolution mode, we ask people if they know someone named something and whose we're searching the address. Every body can reply and not only the one we are looking for. Indeed, information is distributed on every AMI node in the RTs. At the opposite, If we use AR with others criteria, we have no distribution of personal criteria, we ask people if they are following our criteria and only the people matching the criteria can reply. In both cases, we forward packets to others peers if we don't have any reply to an incoming AR.
Note that AR are initiated by over-protocols and sent to AMI engine using an AMI internal service (see Address Request).

AR packet format

Variables:

Name	Meaning
putr	AR transmitter public key. The transmitter is the AR initiator peer.
altr	AR transmitter proposed alias
iptr	AR transmitter IP address. Format: *IP:Port*
pufo	AR forwarder public key, just forwards the request. If current peer is the transmitter, pufo=putr
alfo	AR forwarder alias
criteria	This criteria is protocol-independent and describes criteria a peer has to match to be retrieved. Most of the time, protocol used is 'ami' and the criteria is the public key of the peer whose we want the ip address (Address Resolution mode).
ttl	Time To Live: hops number between peers before packet drop. Default value: 3
arid	AR ID, used to trace and drop looping packets. One arid identifies only information {criteria} but not information {putr,iptr,altr,pufo,alfo}. We can find the same arid for two AR with different transmitters or forwarders but with the same {criteria}.

Note:
We don't use any checksum for AR like in IAI. Indeed, checksums in IAI allows to trace packets and information we already received, but we can receive the same AR more than once.
Packet format:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature' >
 <ar putr='value'
     altr='value'
      iptr='value'
      pufo='value'
      alfo='value'
      ttl='value'
      arid='value' >
  <criteria protocol='protocol name'>
   <param name='parameter name' value='parameter value' and='0 or 1'/>
  </criteria>
 </ar>
</amipacket>
```

If AR is used in Address Resolution mode, criteria is :

```
  <criteria protocol='ami'>
   <param name='pu' value='searched public key' and='1'/>
  </criteria>
```

Otherwise, criteria could be:

```
  <criteria protocol='amirc'>
   <param name='interest' value='cinema' and='1'/>
   <param name='forum' value='art' and='1'/>
  </criteria>
  <criteria protocol='another protocol'>
   <param name='my parameter' value='my value' and='0'/>
  </criteria>
```

Criteria rules:

- MAY contain zero or one Address Resolution criteria with protocol ami and parameter pu. If an AR packet contains one *ami* criteria, it can't contain any other criteria.
- MUST contain one or more mandatory criteria.
- MAY contain zero or more optional criteria.

DTD (validated):

```
<!ELEMENT amipacket (amiping | (iai|ar)+) >
<!ATTLIST ami_packet
   ami_version CDATA #REQUIRED
   reply_port CDATA #REQUIRED
   sign CDATA #REQUIRED >
<!ELEMENT ar (criteria+) >
<!ATTLIST ar
   putr CDATA #REQUIRED
   altr CDATA #REQUIRED
   iptr CDATA #REQUIRED
   pufo CDATA #REQUIRED
   alfo CDATA #REQUIRED
   ttl CDATA #REQUIRED
   arid CDATA #REQUIRED >
<!ELEMENT criteria (param+) >
<!ATTLIST criteria protocol CDATA #REQUIRED >
<!ELEMENT param EMPTY>
<!ATTLIST param
   name CDATA #REQUIRED
   value CDATA #REQUIRED
   and (1|0) '1' >
```

Notes:

<param> tag in AR is a bit different from <param> tag in user.xml : it can't contain sub-elements and attribute *value* is mandatory. Moreover, <param> tag in AR has another attribute : *and* which sets if this parameter is mandatory or not in the search.

Attributes are:

- Name (mandatory) : *param_name*. It is name of the parameter for this protocol.

- Value (mandatory) : *value*. Value of the parameter.

- Mandatory flag (boolean, true by default) : *and*. If *and* attribute is true, it means the peer replying MUST match this parameter. At least one parameter in an AR must be mandatory.

AR transmission

variables:
ttl: time to live (int)
ar : intermediate ar
ais.max : maximum number of replies. Specified by over-protocol when transmitting the AR AIS.

```
begin
 add_search(ar.arid,ais.max)
 ar <- new AR with
  putr= <own PU>
  altr = <own alias>
  iptr = <own IP>
  paufo = <own PU>
  alfo = <own alias>
  criteria = <my criteria>
  ttl = <specified TTL>
  arid = <new ID>
 query_log(arid)
 for each peer in {rt(*) -ar.putr } sorted by peer.pf do
  if check_peer(peer) = true and result_table(ar.arid).retrieved <
result_table(ar.arid).max then
   wait AR_SEND_INTERVAL
   send(ar,rt(peer.pu).ip)
  end if
 end for
end
```

Notes:

- Search deepness is controlled by TTL. Maximum TTL can be 5 (to be studied). Caution: a big TTL will generate huge packet exchanges.

- AIS.max is maximum number of results we want for this AR. It is specified by over-protocol in AR AIS (see Address Request).

We send an AR to find peers matching our criteria. We send the request with our public key, our IP address, our alias, one or more criteria, an ID and the time to live which specifies search deepness. The request is sent to every peer in the routing table sorted by perf. We wait some time between two transmission (time defined in AR_SENDING_INTERVAL); this way we can stop to send AR if we get enough replies (max value specified in AR AIS).

AR reception

variables:

b : intermediary boolean, can we reply to the AR ?
newiai: intermediate IAI

```
begin
 if rt(ar.putr) = null then     //---transmitter analyze
  add_peer(ar.putr,ar.altr,ar.iptr)
```

```
 else if rt(ar.putr) <> null and rt(ar.putr).ip <> ar.iptr then
  alter_peer(ar.putr,ar.iptr)
 end if
 if rt(ar.pufo) = null then     //---forwarder analyze
  add_peer(ar.pufo,ar.ipfo,ar.alfo)   (1)
 else if rt(ar.pufo) <> null and rt(ar.pufo).ip <> ar.ipfo then
  alter_peer(ar.pufo,ar.ipfo)
 end if
 b <- true
 if is_log(ar.arid) = false and ar.ttl<6 then   (2)    //body processing
  query_log(ar.arid)
  if ar.criteria_protocol = 'ami' then
   if rt(ar.criteria_pu) <> null and check_peer(ar.criteria_pu) = true then
    newiai <- new IAI with
     putr = <own pu>
     altr = <own alias>
     criteria = <protocol='ami' param='pu' value='ar.criteria_pu'>
     al = rt(ar.criteria_pu).al
     ip = rt(ar.criteria_pu).ip
     arid = ar.arid
     iaics = <new cs>
    send(newiai,{ar.putr,ar.pufo})   (3)
   else
    b <- false
   end if
  else
   if check_criteria(ar.criteria) = true then
    newiai <- new IAI with
     putr = <own pu>
     altr = <own alias>
     criteria = ar matching criteria ( mandatory and optionnal ones )
     al = <own al>
     ip = <void>
     arid = ar.arid
     iaics = <new cs>
    send(newiai,{ar.putr,ar.pufo})   (3)
   else
    b <- false
   end if
  end if
  if b = false then
   if ar.ttl -1 > 0 then
    add_search(ar.arid,-1)
    newiai <- new IAI with
      putr= ar.putr
      altr = ar.altr
      iptr = ar.iptr
      pufo = <own pu>
      alfo = <own alias>
      criteria = <ar criteria>
      ttl = ar.ttl - 1
      arid = ar.arid
    for peer in {rt(*) - rt(ar.putr) -rt(ar.pufo)} sorted by pf do
     send(newiai,rt(peer.pu).ip)
    end for
   end if
  end if
 end if
end
```

(1) ipfo is got by socket properties
(2) Condition ar.ttl<6 allows to drop cracked packet affecting AMI engine performances. Indeed no AR can have a TTL > 5
(3) See IAI sending : Transmission as reply to an address request (type 2)

When we received an AR there are two cases:

- *we know the reply and if so, we send reply to request transmitter and to last request massager.*
- *we don't. The time to live of address request is decremented. if TTL is more than zero, we ask to every peer we know (except people asking) sorted by performance.*

User criteria

Each peer stores its protocols information in its xml file *user.xml* and relative files. When someone send an Address Request broadcast, the AMI engine read this file to establish if we match the criteria or not. This file is used by all AMI protocols - already existing or not - and also by the Address Resolution function of AMI whose criteria is public key. Note that tags describing the AMI protocol itself are mandatory.

Description files

When AMI runs, it can manage several external protocols. When a user installs a new AMI protocol, specific entries are stored into a special description file with '*.ami*' extension (for io access performances and security reasons) and *user.xml* imports these files as entities. The AMI protocol is self-described in a file called *core.ami*. For example, if you want to add a new protocol, let's say 'amirc' to your configuration, the system will import and write data in a file called *amirc.ami*. The AMI engine will add an entry for *amirc* protocol in file *user.xml.*

AMI service *protocolmanager* (see protocolmanager) must be invoked to add or remove such entries in AMI system.

user.xml format

Note: In this example, protocol *amirc* is given as an example but protocol *ami* is mandatory.

Mandatory AMI parameters:

Name	Meaning
pu	own public key. Couple pu/pr is generated with an AMI external function (see ami engine arguments).
pr	own private key.
al	own alias which will be given to others peers.
All AMI parameters alterable by service *amiparameter*	See list and meanings at: amiparameter

Format:

File **user.xml**:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<user ami_version='0.1' xmlns='http://www.ami.org/user'>
<!entity ami SYSTEM 'core.ami' >
```

```
<!entity amirc SYSTEM 'amirc.ami' >
<protocol name='ami' secure='true' available='true'>
&ami;
</protocol>
<protocol name='amirc' secure='false' available='true'>
&amirc;
</protocol>
</user>
```

File **core.ami** (sample parameters):

```
<configuration>
 <descriptions>
  <protocol_description>
   <note lang='en' value='AMI core protocol'/>
  </protocol_description>
  <param_description param='PU' type='string'>
   <note lang='en' value='Peer public key' />
  </param_description>
 </descriptions>
 <parameters>
  <param name='SYSTEM' value='core.ami' />
  <param name='PU' value='my public key' />
  <param name='AL' value='my default alias' />
  <param name='PR' value='my private key' />
  <param name='TTL' value='3' />
 </parameters>
</configuration>
```

File **amirc.ami** (example of protocol description):

DTD (validated):

```
<configuration>
 <descriptions>
  <protocol_description>
   <note lang='en' value='AMI-based IRC protocol'/>
  </protocol_description>
  <param_description param='SYSTEM' type='string' minlength='1' maxlength='512'
default='amirc.ami'>
   <note lang='en' value='Name of .ami file'/>
  </param_description>
  <param_description param='irc_group_id' type='int' min='0' max='10000'>
   <note lang='en' value='irc group id'/>
  </param_description>
  <param_description param='interest' type='string'>
   <note lang='en' value='subject interest'/>
  </param_description>
 </descriptions>
 <parameters>
  <param name='SYSTEM' value='amirc.ami' />
  <set name='irc_group_id'>
   <entry value='1234'/>
   <entry value='86'/>
  </set>
  <set name='interest'>
   <entry value='cinema'/>
   <entry value='sport'/>
```

```
  </set>
 </parameters>
</configuration>

<!ELEMENT user (protocol+) >
<!ATTLIST user
  ami_version CDATA #REQUIRED
  xmlns CDATA #REQUIRED>
<!ELEMENT protocol (configuration) >
<!ATTLIST protocol
  name CDATA #REQUIRED
  secure (true|false) #REQUIRED
  available (true|false) #REQUIRED >
<!ELEMENT configuration (descriptions?,parameters)>
<!ELEMENT descriptions (protocol_description,param_description*)>
<!ELEMENT parameters (param+,set*)>
<!ELEMENT set (entry+)>
<!ATTLIST set name CDATA #REQUIRED>
<!ELEMENT entry EMPTY>
<!ATTLIST entry value CDATA #REQUIRED>
<!ELEMENT protocol_description (note*) >
<!ELEMENT param_description (note*) >
<!ELEMENT note EMPTY >
<!ATTLIST param_description
  param CDATA #REQUIRED
  type (int|string|boolean|float) #REQUIRED
  min CDATA #IMPLIED
  max CDATA #IMPLIED
  minlength CDATA #IMPLIED
  maxlength CDATA #IMPLIED
  default CDATA #IMPLIED>
<!ATTLIST note
  lang CDATA #REQUIRED
  value CDATA #REQUIRED>
<!ELEMENT param EMPTY >
<!ATTLIST param
  name CDATA #REQUIRED
  value CDATA #REQUIRED >
```

Comments:

- Protocol named 'ami' is mandatory. The number of parameters is not fixed but we must find some mandatory parameters (see table before). Note that the description file *core.ami* will be encoded with a symmetrical key that the user must enter at the AMI engine start.

- Each protocol must contain a mandatory name and we can specify if we want this protocol to be secured or not (by default, is not). If secure='true', every string after the tag <protocol> and before the tag </protocol> (basically, all the description file) will be encoded with the k symmetrical key (blowfish algorithm), to avoid that someone could read protocols parameters just editing the file.

- Attribute *available* informs AMI engine about status of a protocol : in use or not. The user can use one protocol and disable others if he doesn't use them.

- <param> elements must contain a mandatory name and a mandatory value. They can have only one value for the same parameter name. <set> elements are special parameters whish are designed to map several values and can be used to store multiple items like in amirc exemple bellow. Each protocol has a mandatory *SYSTEM* parameter whish gives the description file. This is required because during XML parsing, every ami file (after being decoded if needed) is included inside the main user.xml file and we then lost any file reference wish will be useful when committing some changes in the ami file.

- <param> tag also exist in AR packets and is different in this scope. It is the reason why *user.xml* namespace is different from others AMI formats: *xmlns='http://www.ami.org/user'*.

check_criteria function

If we receive an address request with some application-specific criteria (People wanting to speak about cinema for instance), the AMI engine has to check in our criteria list in our *user.xml* file if we match this request. We use then *boolean check_criteria(criteria)* function. Note that we can use *check_criteria* function only to process non Address Resolution criteria (not based on protocol 'ami'). The AR contains the criteria under this form (see chapter Address request (AR) specifications) :

```
<criteria protocol='protocol name'>
   <param name='parameter name' value='parameter value' and='1 or 0' />
</criteria>
```

Example:

```
<criteria protocol='amirc'>
 <param name='interests' value='cooking' and='1' />
 <param name='interests' value='cake cooking' and='0' />
</criteria>
<criteria protocol='amiskill'>
 <param name='skill' value='cooker' and='1' />
 <param name='location' value='london' and='1' />
</criteria>
```

This request means ' I want to speak with people about cooking and optionally about cake cooking (amirc protocol) and who have cooker skills (eventually to give some lessons) and living in London (amiskill protocol)'. Note that in this case, we are doing a criteria joint over two protocols.

Variables:
criteria : searched criteria as included in the AR
protocol_list : array containing protocols we know and being available (ready to be used)

```
begin
 for each criteria do
  if criteria_protocol in {protocol_list} then
   for each param in {AR <param> tags} do
    if param_and = 1 then
     for each parameter value in user.xml for this protocol and this parameter
do
      if param_value doesn't match a corresponding entry in user.xml then
       return false
      end if
     end for
    end if
   end for
  end if
 end for
 return true
end
```

Goal is to set if we match an AR criteria. For each <criteria> tag in the AR, we first check if this criteria is associated with an existing and in use protocol. If so, we analyze every <param> tag in this criteria. Each <param> tag contains a name and a value. If the parameter is mandatory and that there is no entry in user.xml with a matching param:value, no need to continue, we just return false. We don't check validity of non-mandatory parameters because every AR must contain at least one mandatory parameter and condition 'all mandatory parameters are matching' is enough to return

true.

Protocol description

AMI must be able to deal with any further protocol and requires these protocol to be compliant and follow some rules. An AMI protocol is almost AMI-independent but must at least provide a description file (*.ami* file) wish gives parameters used by AMI engine to check if the current node matches an AR criteria. As we described it before, a parameter is composed of a name and a value (we can find n parameters with the same name but different values).

Parameter description

This is an advanced functionality of protocol description file. The goal is to describe each parameter and set type to limit typing errors when entering parameters. The description tags are located inside the *.ami* file.

Format:

```
<descriptions>
   <description param='value' type='value' min='value' max='value'
minlength='value' maxlength='value' default='value'>
      <note lang='value' value='value' />
   </description>
</descriptions>
```

Description of fields:

Name	Goal
description	A non-mandatory description of a parameter.
param	Name of the parameter described
type	Parameter type: int (integer), string , boolean or float
min	Minimum value if it is a number (int or float)
max	Maximum value if it is a number (int or float)
minlength	Minimum size in characters if it is a string
maxlength	Maximum size in characters if it is a string
default	Default value of this parameter
note	A Note used to be displayed in a configuration tool, can contain several values , one for each language.
lang	Language local of the description (ex : en for English, fr for French, sp for Spanish ...)
value	The note itself in the given language.

AMI Internal Services protocol (AIS)

Definition

AMI over-protocols communicate with AMI engine via a specific protocol : AIS (Internal AMI Services) based on XML. Communication is done with filestreams. This way, any program able to write or read a XML file and parse it, written in any language can become an AMI protocol. If an AMI protocol needs to send an AMI request (an AR for instance), it has to write in the local file *in.xml* which will be processed by AMI engine and result has to be parsed from file *<query_id>.xml*. AIS queries are processed in priority and before standard network AMI queries. AIS queries are encapsulated into an AIS packet (see format bellow).
Variables:

Name	Goal	Mandatory
name	Type of request : *ar* for example	yes
id	Packet id	yes
arg	Argument for query, most of the time is the target of the query	no
value	Needed by some arguments : value to be processed	no
access	Access type (see table below)	yes
max	Maximum number of replies for a query, if not specified, max=-1 (infinite)	no
check	Required by many AIS to process low-level queries or change AMI engine parameters. For AIS queries , check is a trivial hashcode(see hashcode below) encoded with AMI symmetrical key (ami passphrase). For AIS replies, check is the AIS ID encoded with AMI symmetrical key.	no

Format:

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE ais_packet SYSTEM 'ais_query.dtd' >
<ais_packet xmlns='http://www.ami.org/ais' >
 <ais name='query name' id='packet id' arg='query parameter'
value='a value' access='access type' max='max reply number'
check='check value' >
 </ais>
</ais_packet>
```

An AIS can contain only one sub-query like AR or EAI.

Hashcode:
The string used to make the hash is the complete AIS string with check value = '' (and not the actual value which is beeing computed).

DTD (validated):

```
<!-- DTD used for Ami Internal Services queries, do not edit -->
<!ELEMENT ais_packet (ais+)>
<!ATTLIST ais_packet xmlns CDATA #REQUIRED >
<!ELEMENT ais ( ar | eai )? >
```

```
<!ATTLIST ais
   name CDATA #REQUIRED
   id CDATA #REQUIRED
   arg CDATA #IMPLIED
   value CDATA #IMPLIED
   access CDATA #REQUIRED
   max CDATA #IMPLIED
   check CDATA #IMPLIED >
<!-- AR part -->
<!ELEMENT ar (criteria+) >
<!ELEMENT criteria (param+) >
<!ATTLIST criteria protocol CDATA #REQUIRED >
<!ELEMENT param EMPTY >
<!ATTLIST param
   name CDATA #REQUIRED
   value CDATA #REQUIRED
   and CDATA #REQUIRED >
<!-- EAI part -->
<!ELEMENT eai EMPTY>
<!ATTLIST eai
   pu CDATA #IMPLIED
   al CDATA #REQUIRED
   ip CDATA #REQUIRED
   cm CDATA #IMPLIED >
```

The DTD is stored in file *ais_query.dtd*

Reply

Replies from AMI engine are stored in a file named *<id>.xml* (*1234.xml* for example). Every query has its own out file. Reply is wrote under this format:

Format:

```
<ais id='id number' check='value'>
 <reply> reply depending on AIS type</reply>
</ais>
```

DTD (validated):

```
<!-- DTD used Ami Internal Services replies, do not edit -->
<!ELEMENT ais (reply) >
<!ATTLIST ais
   id CDATA #REQUIRED
   check CDATA #REQUIRED >
<!ELEMENT reply (peer*) >
<!ATTLIST ais code CDATA #IMPLIED>
<!ELEMENT peer EMPTY >
<!ATTLIST peer
   pu CDATA #IMPLIED
   al CDATA #IMPLIED
   ip CDATA #IMPLIED >
```

The DTD is stored in file *ais_reply.dtd*

Note that *check* parameter is always needed in replies from AMI engine to authenticate transaction replies.

Access types

Name	Goal
get	Query goal is to *get* something from AMI engine. For example, to get an IP address from a PU.
set	Query goal is to *set* something in AMI engine. For example to change an AMI engine parameter.
add	Add a query in AMI spool. For example, adding an Address Request.
remove	Remove a query from AMI spool. Cancel this query if process is not yet started.

Available AIS

Available services summary:

Access	Name	arg	value	check required ?
get	isolated	-	-	no
	infopeer	pu | al | ip	value of pu or al or ip	no
	amiparameter	parameter to get as <protocol>.<param>	-	no
	protocolstatus	protocol name	-	no
	amiset	Set name as <protocol>.<set>	-	no
set	amiparameter	parameter to set as <protocol>.<param>	value of parameter	yes
	protocolstatus	protocol name	true or false	no
add	ar	-	-	yes
	eai	-	-	yes
	protocolmanager	protocol name	description file path	yes
	cleanrt	older	number of days	yes
		level	[1,3]	yes
		number	[1,-1 (infinite)]	yes
	amiset	destination set as <protocol>.<set>	value to add in the set	yes
remove	ar	-	-	yes
	protocolmanager	protocol name	-	yes
	amiset	destination set as <protocol>.<set>	value to remove from the set	yes

Address Request

Often, some protocols need to operate some address request to find peer lists. In this case, protocol has to provide an AIS to AMI engine.

Rules:

Check required, access type: add,remove

Add Query Format:

```
<ais name='ar' id='value' access='add' max='value like 10' check='value' >
 <ar>
  <criteria protocol='protocol name'>
   <param name='parameter name' value='parameter value' and='0 or 1'/>
  </criteria>
 </ar>
</ais>
```

(Can contain several criteria and several parameters in each criteria).

Note:

The random AIS *id* value will be also be used as AR ID in the AR packet.

Then, AMI engine will process request and will fill the out file with an XML packet like:

Add Reply Format:

```
<ais id='value' check='value' >
 <reply>
  <peer pu='value'
        al='value'
        ip='value' />
  <peer pu='value'
        al='value'
        ip='value' />
  ...
 </reply>
</ais>
```

AR reply has to contain (pu,al,ip) information about found peers.

or if an error occurs:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

Remove Query Format:

```
<ais name='ar' id='value' access='remove' check='value' />
```

Then, AMI engine will process request and will fill out file with an XML packet like:

Remove Reply Format:

```
<ais id='value' check='value'>
 <reply code='code' />
</ais>
```

(code=0 if operation successful)

Code meaning:

Code	Meaning
0	OK, AR canceled
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error

External Address information

An EAI allows a user to enter RT information 'manually'. This information comes from an external information source. It is useful in the following cases:

- **At the very first session:** the External Address Information is the first data filled up in the routing table.
- **Isolated peer :** if no peer is ami-pingable. The AMI protocol needs at least one alive connection.

EAI are sent via an AMI internal service. This way, we avoid mixing EAI with incoming distant AMI queries and we greatly accelerate processing of the EAI in RT. When the AMI engine receive an EAI via AIS, it pushes it directly in a first priority buffer.

'pu' attribute is not mandatory. If it is not given, an amiping is done at given 'ip' address to get it. It requires however for the new peer to be pingable (connected). If pu is given, no real amiping is performed.

Rules:

Check required, access type: add

Variables:

Name	Meaning
pu (optional)	New user public key
al	New user alias
ip	New user IP address at creation. Format: *IP:Port*
cm (optional)	Comments about new user (String(256))

Query Format:

```
<ais name='eai' id='value' access='add' check='value'>
 <eai pu='value' al='value' ip='value' cm='value' />
</ais>
```

Then, AMI engine will process request and will fill the out file with an XML packet like:

Reply Format:

```
<ais id='value' check='value'>
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
0	OK, EAI correctly altered or added
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error

isolated

Rules:

Check not required, access type: get

This AIS is used by external protocol to know if we are in a total isolation case (none RT entry is amipingable). In this case, the protocol could ask for a mandatory EAI for example.

Query Format:

```
<ais name='isolated' id='value' access='get' />
```

Then, AMI engine will process request and will fill out file with an XML packet like:

Reply Format:

```
<ais id='value' check='value'>
 <reply code='true | false | code' />
</ais>
```

Code meaning:

Code	Meaning
2	Problem : wrong packet format
3	Problem : internal error

amiparameter

It is a set of accessors to AMI core parameters like TTL, timeouts...(see list below). This AIS is also deigned as accessor to any AMI protocol.

Rules:

Check required only for set, access type: set, get

'get' query format:

```
<ais name='amiparameter' id='value' arg='protocol.parameter' access='get'/>
```

with 'protocol', the name of the targeted protocol and 'parameter', the targeted parameter for this protocol.

'get' reply format:

```
<ais id='value' check='value' >
 <reply code='value | code' />
</ais>
```

If there is a get error, reply contains an error code (see code meaning below).

'set' query format:

```
<ais name='protocol' id='value' check='value' arg='protocol.parameter'
value='value' access='set' />
```

with 'protocol', the name of the targeted protocol and 'parameter', the targeted parameter for this protocol.

'set' reply format:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
0	OK : amiparameter correctly altered
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : value out of range
5	Problem : unknown parameter

List of AMI core parameters :

Name	Description	Default value	Allowed values
LOCAL	Language in use by AMI	{en]	String [2,5]
TTL	Time To Live, used in AR.	[3]	[0,5]
MAX_RT_SIZE	Maximum number of entries in the RT before a clean level 1 (see clean function)	1E+9 (infinite)	[50, 1E+9]

Name	Description	Default value	Allowed values
NB_SENDER	Number of threads sending AI packets.	[10]	[1,10000]
NB_READER	Number of threads processing incoming AI packets.	[10]	[1,10000]
MAX_OUTSPOOL_SIZE	Maximum size of buffer of packets to be send before blocking the adding of new queries.	[100]	[10, 10000]
MAX_INPUTSPOOL_SIZE	Maximum size of buffer of incoming packets before blocking reading of new incoming packets.	[100]	[10, 10000]
MAX_SEND_TRIES	Maximum number of tries to transmit an AMI packet to a peer before dropping the packet..	[3]	[1,5]
MAX_CHECK_INTERVAL	Trust time of an entry in the RT in seconds (see *dlc* column in RT)	[60]	[1,1E+9(infinite)]
MAX_SEND_TIME	Max sending time for a packet in ms	[10000]	[100,1E+9(infinite)]
CHECK_INTERVAL	General use interval in ms, can be modified following machine performances	[1000]	[10,1E+9]
AMI_PORT	Port used by the AMI listener (AMIReader)	[1139]	[0,65535]
AMIPING_TIMEOUT	Maximum time to get answer from a peer (ms)	[10000]	[100,1E+9]
AR_SEND_INTERVAL	Interval between two AR transmission (ms)	[1000]	[0,1E+9]
BLACK_LIST_LIMIT	Performance point from a peer become automatically black-listed	[-100]	[-1E+9 (infinite), -2],
VERBOSITY	Verbosity level of AMI. 0: full verbosity 1: medium verbosity 2: important messages only 3: none	[2]	[0,3]
MAX_PACKET_SIZE	Maximum number of queries inside a packet to be sent or received.	[20]	[1,20]
AL	Own alias	['default alias']	[String (16)]

Note that this list will probably grow or be modified with AMI versions.

amiset

Amiset AIS is used by protocols to manage a set of data. For instance, it can be used to maintain an address book. AMI supports two type of data storage about a protocol : parameters which are unique and are used to configure the protocol and set which are used to store business data. Note that AMI core protocol doesn't use this AIS because it doesn't have any set.

Rules:

Check required for add and remove,

access type: add, remove, get

'add' query format:

```
<ais name='amiset' id='value' check='value' arg='protocol.amiset' value='value
to add' access='add'/>
```

with 'protocol', the name of the targeted protocol and 'amiset', the targeted set for this protocol.

'add' reply format:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

If there is a get error, reply contains an error code (see code meaning below).

'remove' query format:

```
<ais name='amiset' id='value' check='value' arg='protocol.amiset' value='value
to remove' access='remove'/>
```

with 'protocol', the name of the targeted protocol and 'amiset', the targeted set for this protocol.

'remove' reply format:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

'get' query format:

```
<ais name='amiset' id='value' check='value' arg='protocol.amiset'
access='get'/>
```

with 'protocol', the name of the targeted protocol and 'amiset', the targeted set for this protocol.

'get' reply format:

if operation completed:

```
<ais id='value' check='value' >
 <reply>
  <entry value='value1'/>
  <entry value='value2'/>
  <entry value='value3'/>
  ...
 </reply>
</ais>
```

if operation fails:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

If there is a get error, reply contains an error code (see code meaning below).

Code meaning:

Code	Meaning
0	OK : amiset correctly altered
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : value out of range
5	Problem : unknown set

infopeer

This request is used to get information about an entry in RT. You have to give either pu or ip or al and AMI engine returns corresponding peer(s) matching your query.

Rules:

Check required, access type: get

Query format:

```
<ais name='infopeer' id='value' check='value' arg='pu | ip | al' value='value'
access='get' />
```

Reply format:

If arg='pu', reply will contain <al> and <ip>
If arg='ip', reply will contain <pu> and <al>
If arg='al', reply will contain <pu> and <ip>

```
<ais id='value' check='value' >
 <reply>
  <peer pu='value'
        al='value'
        ip='value' />
  <peer pu='value'
        al='value'
        ip='value' />
  ...
 </reply>
</ais>
```

or if an error occurs:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : unknown argument
5	Problem : No result

If arg='pu', there is always one unique reply.

If arg='ip' or 'al', it may have several replies.

cleanrt

It is used to clean old and useless entries in RT.

Rules:

Check required, access type: add

Query format:

```
<ais name='cleanrt' id='value' check='value' arg='older' value='value in seconds' access='add' />
```

or

```
<ais name='cleanrt' id='value' check='value' arg='level' value='level value' access='add' />
```

or

```
<ais name='cleanrt' id='value' check='value' arg='number' value='minimum number of peers to be dropped' access='add' />
```

Reply format:

if success,

```
<ais id='value' check='value' >
 <reply code='number of peers dropped' />
</ais>
```

if any error occurs,

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

Argument description:

- *older* argument is used to specify *dlc* date in seconds from peers are dropped. For example, if *older*=60, every peers which date last check is greater than 1 min will be dropped. Caution : if *older* value is lower than *MAX_CHECK_INTERVAL* (see amiparameter), you can drop trusted peers.

- *level* argument provides generic cleanup scenarios with three levels : level 1: every peer which dlc is greater than 10 days are dropped ; level 2: Every peer which dlc is greater than MAX_CHECK_INTERVAL and which pf is lower than 10 are dropped ; level 3 : Every peer which dlc is greater than MAX_CHECK_INTERVAL are dropped.

- *number* argument provides possibility specify a fixed number of peers to be dropped. *number* peers are dropped, even if they are trusted (in this case, they are dropped from lowest pf to highest pf).

Code meaning:

Code	Meaning
0	OK, RT cleaned (The number of erased peers is written after the '0' and a slash)
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : out of range

protocolstatus

It is used to get or set status of a given protocol (available or not). If a protocol is not available, we will not match a broadcast dealing with this protocol. All protocols status are in *user.xml* file (and relative files), see user.xml format .

Rules:

Check not required, access type: get, set

get query format:

```
<ais name='protocolstatus' id='value' arg='protocol' access='get' />
```

get reply format:

```
<ais id='value' check='value' >
 <reply code='true | false | code' />
</ais>
```

If an error occurs (because of a wrong protocol name for example), a code is returned (see code table below).

set query format:

```
<ais name='protocolstatus' id='value' arg='protocol' value='true|false'
access='set' />
```

set reply format:

```
<ais id='value' check='value' >
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
0	OK : protocol status correctly altered
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : protocol does not exit

protocolmanager

This is a core service used to add or remove entries in *user.xml* file in order to plug or unplug AMI protocols to the system.

Rules:

Check required, access type: add,remove

Query Format for adding:

```
<ais name='protocolmanager' id='value' arg='protocol name/(true|false)'
value='description file location' access='add' check='value'/>
```

- arg contains the protocol name followed by a slash and a boolean representing security policy of the protocol (encoded or not).

Query Format for removing:

```
<ais name='protocolmanager' id='value' arg='protocol name' access='remove'
check='value' />
```

Then, AMI engine will process request and will fill the out file with an XML packet like:

Reply Format:

```
<ais id='value' check='value'>
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
0	OK, protocol correctly added or removed
1	Problem : check failed
2	Problem : wrong packet format
3	Problem : internal error
4	Problem : Description file not found
5	Problem : Incorrect description file.

amistop

This is a core service used to stop properly the ami engine.

Rules:

Check not required, access type: add

Query Format:

```
<ais name='amistop' id='value' access='add' />
```

Then, AMI engine will process request and will fill the out file with an XML packet like:

Reply Format:

```
<ais id='value' check='value'>
 <reply code='code' />
</ais>
```

Code meaning:

Code	Meaning
0	OK, ami engine is about to stop
2	Problem : wrong packet format
3	Problem : internal error

AIS engine

Here is described how AMI engine processes AMI internal services queries. A specific thread read recursively file *in.xml* to detect AIS adding event. Note that timeouts are managed by over-protocols. When maximum time for the AIS is reached, the over-protocol should send a remove access type AIS to AMI for cleanup. In this algorithm, we just describe AR processing ; others AIS queries are trivial and described previously.

When receiving an AR AIS:

```
begin
 add_search(ais.id,ais.max)
 send the AR
end
```

We create a new entry in result_table for this AR, identified by its ID.

When receiving a reply for this AR (a new result in result_table has been detected):

```
begin
 if first reply for this id then
  write in out file:
  <ais id='value' check='value' >
  <reply>
   <peer pu='pu1'
         al='al1'
         ip='ip1' />
  </reply>
  </ais>

 else
  add a new reply in out file:
   <ais id='value' check='value'>
    <reply>
     <peer pu='pu1'
         al='al1'
         ip='ip1' />
   <peer pu='pu2'
         al='al2'
         ip='ip2' />
    </reply>
   </ais>
 end if
 if result_table(iai.arid).retrieved = result_table(iai.arid).max then
   del_search(ais.arid)
 end if
end
```

If we get some result for a waited AR and we haven't reach maximum number of results, we alter the out file. Note that we alter file at each result because this way, over-protocol get immediate results.

Packets compatibility matrix

Available mixes

This array gives you request mixes we can use inside an ami packet.
For instance, you can process a packet like:

```
<amipacket ami_version='0.1' reply_port='port' sign='signature'>
 <ar>
 ...
 </ar>
 <iai>
 ...
 </iai>
```

```
</amipacket>
```

	IAI	AR	AMIPING
IAI	yes	yes	no
AR	yes	yes	no
AMIPING	no	no	no

Global DTD

This DTD is a summary of all packet-types DTD (AMIping, IAI and AR) and is used for incoming packet validation.

```
<!--packet part-->
<!ELEMENT amipacket (amiping | (iai|ar)+) >
<!ATTLIST ami_packet
   ami_version CDATA #REQUIRED
   reply_port CDATA #REQUIRED
   sign CDATA #IMPLIED >
<!--criteria part-->
<!ELEMENT criteria (param+) >
<!ATTLIST criteria protocol CDATA #REQUIRED >
<!ELEMENT param EMPTY>
<!ATTLIST param
   name CDATA #REQUIRED
   value CDATA #REQUIRED
   and (true|false) 'true' >
<!--amiping part-->
<!ELEMENT amiping EMPTY >
<!ATTLIST amiping pu CDATA #IMPLIED >
<!--iai part-->
<!ELEMENT iai (criteria+) >
<!ATTLIST iai
   putr CDATA #REQUIRED
   altr CDATA #REQUIRED
   al CDATA #REQUIRED
   ip CDATA #REQUIRED
   arid CDATA #REQUIRED
   iaics CDATA #REQUIRED >
<!--ar part-->
<!ELEMENT ar (criteria+) >
<!ATTLIST ar
   putr CDATA #REQUIRED
   altr CDATA #REQUIRED
   iptr CDATA #REQUIRED
   pufo CDATA #REQUIRED
   alfo CDATA #REQUIRED
   ttl CDATA #REQUIRED
   arid CDATA #REQUIRED >
```

SOAP encapsulating

In order to foresee future exchange w3c standards, all packets will be SOAP compliant even if we don't use for the moment real functionality of this norm. SOAP is 'a lightweight protocol for exchange of information in a

decentralized, distributed environment. It is an XML based protocol that consists of four parts: an envelope that defines a framework for describing what is in a message and how to process it, a transport binding framework for exchanging messages using an underlying protocol, a set of encoding rules for expressing instances of application-defined data types and a convention for representing remote procedure calls and responses'. Find SOAP documentation on : http://www.w3.org . As SOAP protocol writing is in process, this part is subject to changes.

To assure SOAP compliance, AMI packets will not be sent directly over TCP protocol but encapsulate in a SOAP envelope (itself encapsulated in a HTTP packet, see next part).

```
<env:Envelope xmlns:env='http://www.w3.org/2001/09/soap-envelope'>
<env:Body>
<amipacket ami_version='0.1' reply_port='port' sign='signature'>
 <iai>
  ...
 </iai>
</amipacket>
</env:Body>
</env:Envelope>
```

HTTP encapsulating

HTTP tunneling

In order to go over a lot of network difficulties (firewall.), we encapsulate the SOAP envelops into HTTP packets (HTTP tunneling). This way, AMI packets will be read as normal HTTP packets. Note than we don't really use HTTP features, goal is not to contact real HTTP servers but only to go through firewalls and proxies. AMI packets are encapsulated in a HTTP request (a POST) with following header:

```
POST /ami HTTP/1.1
Host: hostname
Content-Type: text/xml; charset='utf-8'
Content-Length: body size
SOAPAction: 'http://www.ami.org'
```

Total AMI packet sample

```
------------packet start-----------------

POST /ami HTTP/1.1
Host: 177.34.5.4
Content-Type: text/xml; charset='utf-8'
Content-Length: 298
SOAPAction: 'http://www.ami.org'

<env:Envelope xmlns:env='http://www.w3.org/2001/09/soap-envelope'>
<env:Body>
<amipacket ami_version='0.1' reply_port='port' sign='signature'>
 <iai putr='e2fg5tgEr'
      altr='toto'
      al='titi'
      ip='102.4.8.4'
      arid='0'
      iaics='459482124' >
  <criteria protocol='ami'>
   <param name='pu' value='f45hY7ddh6t'/>
  </criteria>
 </iai>
</amipacket>
```

```
</env:Body>
</env:Envelope>

------------packet end----------------
```

Note we have a blank line after POST header to separate header from body.

Packets alias

To limit packet size, we must use shorter tags than given previously. We keep long-format tags notations for understanding but note the following mapping between regular tags and real-packet tags:

Regular tag	Used tag
amipacket	**ap**
ami_version	**av**
reply_port	**rp**
sign	**s**
amiping	**pg**
pu	**pu**
iai	**ai**
putr	**pt**
altr	**at**
criteria	**c**
protocol	**pl**
param	**pm**
name	**n**
value	**v**
and	**a**
al	**al**
ip	**ip**
arid	**rd**
iaics	**ic**
ar	**ar**
iptr	**it**
pufo	**pf**
alfo	**af**
ttl	**tl**
ami	**ami**

Note that in the same time, we could remove Soap part from packets to save space as it doesn't yet really useful.

AMIPATH obtaining

AMI needs to find its working directory in order to start properly. AMI engine will try to get this information with 4 distinct methods (sorted) :

Method	Ressource
Command line	**'-d *amipath* '** when starting AMI engine.
Environment variable	**$AMIPATH**
Property file	**$HOME/amirc.properties**
Current directory	**'.' directory**

Ami engine external functions

Available options

We can specify some argument in command line. AMI provides following external functions:

Name	Meaning
-d *directory*	Specifies than AMI directory (place where you find files ami.jar, *user.xml*, *rt.xml* and all AMI relative files) is *directory* . If not given, AMI engine check the environment variable AMIPATH and if this variable is void, it looks for an amirc.properties file in $HOME and there is not, it uses current directory.
-install	Forces AMI installation. (if the system already exists, it is overwritten and reinstalled) Can be called with *-d* option. This option must be called only one time for each value of AMIPATH, it will create AMI environment and set password and PKI keys. If not called, AMI will cannot start. When called, AMI will: 1- Create needed files : *user.xml* and *rt.xml* with default or empty values. 2- Ask for an AMI passphrase (symmetrical key). 3- Generate a couple of keys (public key / private key) and put them in *user.xml*. Will NOT override any existing *user.xml* file. Caution: public key identifies the peer. If peer changes its pu, it will be taken as a new peer and will no more be reachable by others peers. Backup *user.xml* file to keep a trace of this key. Moreover, pr (private key) must stay secret to avoid some cracking. Pr is in *user.xml* file encoded with the ami passphrase (k). So you need to: - Backup *user.xml* file - Remind AMI passphrase or symmetrical key (*k*) - Keep *user.xml* out of access.

-chgpass	Called to change the ami passphrase. This passphrase must be at least 8 characters long and should be constituted of several words. It is case sensitive and accept any punctuation character. Example: ' my tailor is rich' or 'Rggt6fg64jU tyh8 1Gre'. But remind that you will need to enter it at each ami start and that it can be required by some AMI over-protocols.
-importrt *=filenam e.xml*	Used to import data of a RT (*rt.xml* file) - that somebody sent to you - in local *rt.xml*. Local RT is updated with this RT. Caution: AMI assumes that data included in filename.xml is firstly on local data : local RT entries are not dropped but entries with same pu than in *filename.xml* will be overwritten. The ami passphrase will be required in order to process the import.
-version	Displays AMI engine version
-nogui	Uses AMI without graphical display (useful under Unix for scripting for instance).
-help	Displays the AMI help message detailing available options.

Usage:
ami [-d=<*directory*>] [-nogui][-install | -chgpass | -version | -help | -importrt=<*filename*>]

To call this functions:
java ami.AMIMain <parameter>
for example:
$ *java ami.AMIMain -install -d=/home/toto*

To start AMI engine:
java ami.AMIMain [-d directory]
For example if ami path is */opt/ami*:
$ *cd /opt/ami*
$ *java ami.AMIMain*
or
$ *export AMIPATH=/opt/ami*
$ *java ami.AMIMain*
or
$ *java ami.AMIMain -d /opt/ami*