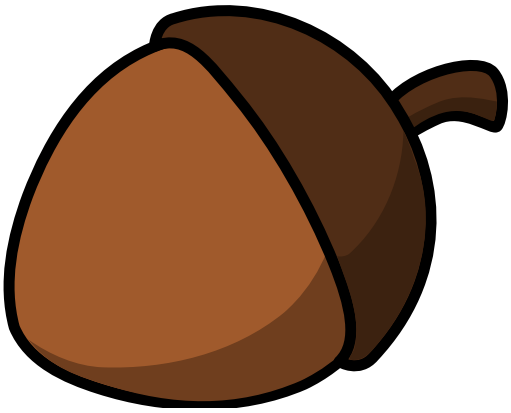


Denobo

Denobo Message Transfer Protocol (DMTP)
Version 1.0.0.0 (Beta)

Preface – Intention

The Denobo project was cooked up in around a month as a second-year university project exploring multi-agent middleware. It's name 'Denobo' is a neologism derived from the phrase "Definitely not Boris" to emphasise its radically different approach to threaded agentware from the software that served as the inspiration for the official Denobo Java library, a piece of agentware named Boris.

This difference lies in the structure of the middleware itself. While other multi-agent middleware libraries emphasise hierarchy and parent-child relationships between agents, Denobo maintains a completely flat network structure and strictly peer-to-peer approach that offers extra redundancy, flexibility and modularity.

This is not for one moment to say that this approach is superior to a more structured hierarchical approach, only that it offers something different, and allows an end-user to use an extremely thin API to set up and use Denobo networks from their own Java software. For instance, on a Denobo network the only way to have an independent subsystem is to have it isolated and not connected to anything else. As soon as an agent is linked in to a Denobo network, it and everything it is connected to is discoverable, and can both send and receive send traffic via the new link.

This protocol specification is intentionally kept programming-language-independent to help serve as a reference guide for those seeking to write their own Denobo-compatible software in a language of their choice.

The intention of DMTP is to provide a means of transmitting serialised message data peer-to-peer between network-enabled actors in the Denobo multi-agent middleware. The protocol supports encryption and compression of packet payloads, and authentication of connecting peers using username-password pairs.

The original Denobo Java library and DMTP specification was written collaboratively by Saul Johnson, Alex Mullen and Lee Oliver (studying under Dr. Mark Truran and Dr. Simon Lynch).

Important: As of this version of DMTP, only basic support is provided for encryption and compression over sockets. Agent names are plainly visible from remote machines and a malicious user could exploit any number of as yet unaddressed network characteristics to intercept and obtain data intended for another recipient. In short, software implementing this iteration of DMTP should not be considered a secure platform or be used for critical systems.

Chapter I – System-Wide Conventions

Agent Naming & Identification

Denobo agents constructed for the purposes of joining a network must abide by certain naming conventions in order to be considered valid. This is in the interest of security (malicious agent names may interfere with packet parsing) and integrity (badly-formed agent names may result in message dropping or in the incorrect recipient receiving and processing the message). Agent names must be:

- Well-Formed:
  - Consisting of only numbers (0-9), letters (a-z, A-Z) and underscore (_) characters, and may not begin with a number.
  - Free from spaces, tabs, null characters, or whitespace of any kind.
  - Use only valid ASCII characters.
  - May not be null or empty.
- Unique:
  - Agent names **must** uniquely identify the agent. Duplicate agent names may result in undesirable behaviour.
  - Because of the manner in which message routing is implemented, the non-uniquely-named agent that takes the least amount of agents to pass through and/or takes the least amount of time to reach will receive the message. Any other agents with the same name will not.
  - The manner in which this uniqueness is ascertained is up to the end user. Using an agent as a central name registry is one option that can trivially be implemented using a basic Denobo network.

The official Denobo middleware will not allow agents with illegal names to be constructed. However, **no checking of any kind is done for duplicate agent names**. The responsibility for this lies with the end user.

Ending a Session

To end a session, either peer should disconnect by closing the socket. There are no formal "session end" codes necessary to perform a session shutdown.

Chapter II – An Overview of Packets

Denobo agents communicate over sockets using token-delimited packets. Each packet begins with a '@' sign (ASCII character 0x40) and ends with a '$' sign (ASCII character 0x24). Between the starting and ending tokens lies a key-value query string of the following format:

name1=value1&name2=value2

These query strings use a URL-style format to escape any characters that may interfere with their parsing. For example, the key-value pair:

{myequation, 1+1=2}

Would become the following query string once formatted for transmission as a packet:

myequation=1%2b1%3d2

It is advised that, as in the official agentware, query strings escape all characters on the value side of a key-value pair that are not letters (A-Z, a-z), numbers (0-9) or underscore (_) characters.

Using this format, it is possible for query strings to be nested. Many types of Denobo packet transmit nested query strings as part of their payload.

Importantly, it is entirely possible for the data between the starting and ending tokens in the socket stream to be encrypted and/or compressed. Good DMTP-compliant agentware should be able to decompress and/or decrypt packets received from a remote peer depending on the metadata provided during handshaking/session modification (see *Chapter VII – Session Flow*). Compression (if any) always takes place **before** encryption.

Specific Packet Fields

At the top level, Denobo packets contain two fields:

- The **code** field, which contains a *packet code* (see *Chapter III - Packet Codes*) identifying the purpose and format of the packet. This code instructs the remote peer on how the packet body should be parsed and interpreted.
- The **body** field, which contains the packet payload. The format of this depends on the packet's code and may be a nested query string.

Chapter III - Packet Codes

Packet codes are transmitted as a sequence of 3 ASCII digits (0-9). They are contained under the ***code*** key of a top-level packet query string and specify a packet's purpose.

1XX - Handshake/Initialisation Codes

Codes 100 – 199 (inclusive) are reserved for packets relating to connection handshaking and the transmission of metadata between connection peers.

100: GREETINGS

Packet code 100 is the GREETINGS code, and is sent from the peer initiating the connection (i.e. the local peer) to the one receiving the connection (the remote peer) to identify itself as a Denobo peer node at the start of a peer-to-peer session.

The packet body contains the name of the local peer in a nested query string under the key ***name*** and the local peer's public key (a large integer) under the key ***pubkey***. Although the public key of the local peer is transmitted as part of the 100 packet, it is still up to the remote peer to decide on whether or not the session will be encrypted and reply with its own public key. The example packet below specifies that the local peer's name is "LocalAgentName" and its public key is 102947287 (real public keys will be much larger).

Example Packet

@code=100&body=name%3dLocalAgentName%26pubkey=102947287$

Possible Responses

The local peer should be prepared for the following responses from the remote peer:

- 101 (ACCEPTED): Remote peer accepted the connection. Session clear to begin.
- 102 (CREDENTIALS_PLZ): Authentication is required to access remote peer. Local peer should reply with a 103 (CREDENTIALS) packet containing a username and password. Remote peer should terminate the connection if a 103 packet is not provided within a timeout.
- 200 (SET_COMPRESSION): Remote peer is specifying that the session will be compressed and is replying with the name of the compression algorithm to use.
- 201 (BEGIN_SECURE): Remote peer is specifying that the session will be encrypted and is replying with its own public key.
- 400 (NO): Remote peer refused the connection without specifying why.
- 401 (TOO_MANY_PEERS): Remote peer refused the connection because its maximum number of peers has been reached.

101: ACCEPTED

Packet code 101 is the ACCEPTED code, and is sent from the remote peer to the local peer in response to a 100 (GREETINGS) or 103 (CREDENTIALS) packet to confirm that the session is clear to begin. Local peers receiving the 101 status code may begin transmitting 3XX packets containing live session data. Handshaking and session modification is considered to have ended after a 101 code is transmitted.

The packet body contains the name of the remote peer in a nested query string under the key ***name***. This packet contains no other data. The example packet below specifies that the remote peer's name is "RemoteAgentName".

Example Packet

@code=101&body=name%3dRemoteAgentName$

Possible Responses

- A packet with any live session code (3XX) or error code (4XX) can be received directly following the sending of a 101 code. No Handshake/Initialisation codes (1XX) or session modification codes (2XX) should be expected as a 101 code signals the end of handshaking and session modification.

102: CREDENTIALS_PLZ

Packet code 102 is the CREDENTIALS_PLZ code, and is sent from the remote peer to the local peer in response to a 100 (GREETINGS) packet to ask for a username and password as an additional authentication measure. The remote peer is preconfigured with a username/password pair to authenticate against, which can be loaded from any data source.

The body of this packet contains no data.

Example Packet

@code=101&body=$

Possible Responses

- 103 (CREDENTIALS): Local peer is offering the username and password contained in the packet body as authentication.
- 402 (NO_CREDENTIALS): Local peer does not have any credentials to offer and has terminated the session.
- No response after a reasonable timeout should be treated as a 402 packet.

103: CREDENTIALS

Packet code 103 is the CREDENTIALS code, and is sent from the local peer to the remote peer in response to a 102 (CREDENTIALS_PLZ) packet to offer up its username and password as authentication. Importantly, the session may or may not be encrypted at this stage, depending on the configuration of the remote peer. **Credentials may end up being sent in the clear**.

The packet body contains the username and password that make up the credentials in a nested query string under the keys ***username*** and ***password*** respectively. The example packet below contains “MyUsername” and “MyPassword” respectively as the username and password.

Example Packet

@code=103&body=username%3dMyUsername%26password%3dMyPassword$

Possible Responses

- 101 (ACCEPTED): Remote peer accepted the connection and session clear to begin.
- 400 (NO): Remote peer refused the connection without specifying why.
- 403 (BAD_CREDENTIALS): Remote peer refused the connection because the credentials provided in the 103 packet were not acceptable.

2XX Session Modification Codes

Codes 200-299 (inclusive) are reserved for packets relating to the modification of the session state and can only be transmitted by the remote peer. The remote peer has ultimate control over modification of the session state before live 3XX packets start to be transmitted. These codes may be sent in direct response to a 100 (GREETINGS) packet, but may no longer be sent once the remote peer has transmitted either a 102 (CREDENTIALS_PLZ) packet or a 101 (ACCEPTED) packet.

200: SET_COMPRESSION

Packet code 200 is the SET_COMPRESSION code. Upon receipt of a 200 code, the local peer should switch the compression algorithm it uses to compress packet payloads to the one specified in the packet body (see *Chapter IV – Compression Techniques*).

The packet body contains the name of the compression algorithm to use in a nested query string under the key ***name***. This packet contains no other data. The example packet below sets the algorithm to LZW compression.

Example Packet

@code=200&body=name%3dlzw$

Possible Responses

- No direct response is expected to a 200 (SET_COMPRESSION) packet.

201: BEGIN_SECURE

Packet code 201 is the BEGIN_SECURE code. Upon receipt of a 201 code, the local peer should calculate the shared secret key using the public key contained in the packet body and begin transmitting encrypted packets. Immediately following the transmission of the 201 packet by the remote peer, the remote peer will have become encrypted (i.e. this is the last unencrypted packet that will be transmitted before the session becomes secure). In the example packet below, the public key is 394837836 (real public keys will be much larger).

Example Packet

@code=201&body=pubkey%3d394837836

Possible Responses

- No direct response is expected to a 201 (BEGIN_SECURE) packet.

3XX Live Session Codes

Codes 300-399 (inclusive) are reserved for packets relating to message transmission, routing and any other live session data. Importantly, 3XX codes can be transmitted and received by either peer. At this point in the session, both peers are equal. There is no 'client' or 'server'.

300: SEND_MESSAGE

Packet code 300 is the SEND_MESSAGE code. The function of a 300 packet is to transmit a serialised message across a socket en-route to its recipient.

In the example packet below, the serialised message data is contained within the packet body. The message is travelling along a route "Agent1->Agent2->SocketAgent1-> SocketAgent2->RemoteDestination" and contains the payload "Hello World!"

Note that the query string nesting in the example is 3 levels deep: the top level packet query string, the message query string within that under the ***body*** key and the route query string within that under the ***route*** key.

See *Chapter VI – Message Structure and Serialisation* for details on how messages are serialised for transmission across sockets.

Example Packet

@code=300&body=id%3d2555bbf8ef7845e0143e290a714fa5ccfb62fea63590d403eff0d4cf1918f8dc%26route%3dposition%253d4%2526route%253dAgent1%25252cAgent2%25252cSocketAgent1%25252cSocketAgent2%25252cRemoteDestination%26data%3dHello%2520World%2521$

Possible Responses

No direct response is expected to a 300 (SEND_MESSAGE) packet.

301: POKE

Packet code 301 is the POKE code, instructing the receiving peer to reply with another 301 packet of its own. The sole purpose of this is to establish whether or not the recipient agent is healthy and responding. This code can also be used as a 'keep-alive' code to keep a connection from timing out.

The body of this packet contains no data.

Example Packet

@code=301&body=$

Possible Responses

- Any other live session code (3XX) or error code (4XX). Pokes are asynchronous, and another packet could arrive before the remote agent pokes us back.
- 301 (POKE): The agent replied to the poke with a poke of its own, indicating that the connection and agent are still healthy.

302: ROUTE_TO

Packet code 302 is the ROUTE_TO code, instructing the peer that receives it that it is required to return the shortest route to a named agent that it may or may not be directly connected to. The route should be returned as a list of agent names that a message must pass through to reach its destination (see *303: ROUTE_FOUND* for more details).

In the example packet below, the recipient agent is being instructed to return the route to an agent called "RemoteDestination". The local route is always transmitted as part of a 302 packet so the remote agent can continue to build it on the remote end. In this instance, the route has already been mapped as "Agent1->Agent2->SocketAgent1" before the routing request crossed the socket, where "SocketAgent1" is the peer on the other side of the connection. See *Chapter VI – Message Structure and Serialisation* for details on how routes are serialised for transmission across sockets.

Note that the query string nesting in the example packet is 3 levels deep: the top level packet query string, the ROUTE_TO packet query string within that under the ***body*** key and the serialised route within that under the ***route*** key.

Example Packet

@code=302&body=localroute%3dposition%253d0%2526route%253dAgent1%25252cAgent2%25252cSocketAgent1%26to%3dRemoteDestination$

Possible Responses

No direct response is expected to a 302 (ROUTE_TO) packet. If a route is found, a 303 (ROUTE_FOUND) packet may be returned asynchronously at any time following the transmission of the 302 packet.

303: ROUTE_FOUND

Packet code 303 is the ROUTE_FOUND code, and may be returned by a peer at any time following receipt of a 302 (ROUTE_TO) packet. This packet should contain a completed route from the originator of a routing request to the destination agent originally named in the 302 packet.

In the example packet below, the route to "RemoteDestination" has been calculated as "Agent1->Agent2-> SocketAgent1-> SocketAgent2->RemoteDestination" and the name of the agent routed to is "RemoteDestination". See *Chapter VI – Message Structure and Serialisation* for details on how routes are serialised for transmission across sockets.

Note that the query string nesting in the example packet is 3 levels deep: the top level packet query string, the ROUTE_FOUND packet query string within that under the ***body*** key and the serialised route within that under the ***route*** key.

Example Packet

@code=303&body=to%3dRemoteDestination%26route%3dposition%253d0%2526route%253dAgent1%25252cAgent2%25252cSocketAgent1%25252cSocketAgent2%25252cRemoteDestination$

Possible Responses

No direct response is expected to a 303 (ROUTE_FOUND) packet. Transmitting this packet may, however, cause a sudden influx of 300 (SEND_MESSAGE) packets as messages waiting for a route to the destination agent are sent over the socket.

304: INVALIDATE_AGENTS

Packet code 304 is the INVALIDATE_AGENTS packet, used to inform a peer that a set of agents have become invalidated (i.e. routes using them may no longer be valid). Peers receiving this code should remove the agents listed in the packet body from any and all routing tables maintained in their environment.

The packet body contains a nesteq query string with two keys. The ***visitedagents*** key contains a semicolon-delimited list of agent names that have already been 'informed' of the agent invalidation and should not need further updating. The ***invalidatedagents*** key contains a semicolon-delimited list of the names of the agents that have become invalidated. The peer receiving the 304 packet should take care to remove these agents from any and all local routing tables to avoid dropping messages heading along routes that no longer exist.

Example Packet

@code=304&body=invalidatedagents%Agent6%253bAgent2%26visitedagents%3dAgent6%253bAgent7%253bAgent8%253bAgent2%253bAgent4$

Possible responses

No direct response is expected to a 304 (INVALIDATE_AGENTS) packet.

4XX Error Codes

Codes 400-499 (inclusive) are reserved for messages relating to errors encountered by either peer in response to a packet they receive. Direct responses are never expected to any of these codes and therefore to the “Possible Responses” section has been omitted.

400: NO

Packet code 400 is the NO code, indicating that a generic error was encountered by the peer. Peers should avoid using this code whenever a more specific error code is available, but **must** transmit an error code of some kind whenever a packet is received that they are unable to deal with. The body of the message may contain additional information about the exact nature of the error.

In the example packet below, a connection has been rejected because the local peer has submitted a name that the remote peer does not recognise as valid.

Example Packet

@code=400&body=Your name doesn%27t look valid to me%2e$

401: TOO_MANY_PEERS

Packet code 401 is the TOO_MANY_PEERS code, indicating that remote peer is configured with a peer limit and that limit has been reached. A 401 code is sent in response to a 100 (GREETINGS) code rejecting the connection on this basis.

The body of this packet contains no data.

Example Packet

@code=401&body=$

402: NO_CREDENTIALS

Packet code 402 is the NO_CREDENTIALS code, indicating that the local peer has not been configured with a username and password and as such is unable to provide the credentials requested. A 402 packet is sent in response to a 102 (CREDENTIALS_PLZ) packet acknowledging that it has no credentials to provide before closing the connection.

The body of this packet contains no data.

Example Packet

@code=402&body=$

403: BAD_CREDENTIALS

The peer is rejecting the username and password passed to it in a 103 (CREDENTIALS) packet because they are incorrect, badly-formed or otherwise unacceptable. The connection will be closed.

The body of this packet contains no data.

Example Packet

@code=403&body=$

Chapter IV – Compression Techniques

Denobo officially supports a number of compression techniques for use with the 200 (SET_COMPRESSION) packet code. This chapter specifies in detail the format of each.

Because the 201 packet is expected to have been transmitted beforehand, it is assumed that both peers know which compression technique is being used in any one session. Therefore, no identification bytes or 'magic numbers' are used to mark compressed data as using one compression technique or another.

It is worth noting that compression has the potential to increase the size of packets unsuitable for compression (i.e. small packets or packets containing high-entropy data) and that compression will greatly increase the overhead associated with packet processing.

Compression Technique: None

The unique identifier for technique is 'none' (case sensitive). This indicates that no compression should be performed on transmitted packets.

Compression Technique: Basic

The unique identifier for this technique is 'basic' (case sensitive). Simple semi-adaptive Huffman coding based on byte frequencies. Compression takes place in stages:

- One pass is done on the data to collect byte frequency values.
- A Huffman tree is generated from these frequencies.
- Each leaf on the Huffman tree is back-traced to the root, giving a prefix code.
- Each prefix code is added as an entry in a lookup table.
- The lookup table is used to translate bytes into prefix codes, which are concatenated to create a compressed set of bytes.
- This set of bytes represents the compressed payload.

Additional metadata must be included alongside the compressed payload in order to enable its decompression by the receiving peer. Included metadata follows:

- A BSD checksum byte of the **uncompressed** data to verify data integrity.
- 4 bytes containing the length of the compressed data **in bits** (the compressed payload is not necessarily a discrete number of bytes in length).
- A serialised representation of the prefix code table (necessary for translation of bit codes back to bytes).

For this reason, basic coding has the potential to dramatically increase the size of packets too small or entropic to be efficiently compressed.

Formal Format Specification

A more detailed description of the format of a compressed BASIC data packet follows:

- **0x00:** BSD checksum byte.
- **0x01-0x04:** Length of compressed data in bits.
- **0x05-N:** Table entries of serialised prefix code table.
    - **Offset 0:** A zero byte (0x00) or a 255 byte (0xFF) if the entry is the last one in the table.
    - **Offset 1:** The value of the byte the prefix code represents.
    - **Offset 2:** The length of the prefix code **in bits**.
    - **Offset 3+**: The prefix code itself.
- **N**-*: The compressed payload.

Compression Technique: LZW

The unique identifier for this technique is 'lzw' (case sensitive). Compression is performed using a 16-bit version of the Lempel-Ziv-Welch algorithm. Compression takes place incrementally. As this is a well-known algorithm that is already specified, it will not be discussed in detail here. What will be discussed, however, is the precise format in which it is transmitted.

As it is possible to decompress LZW-coded data directly from compressed data no dictionary or lookup table of any kind is included with the transmitted payload. The receiving peer is expected to incrementally reconstruct the encoding dictionary directly from the compressed data stream.

For the sake of data integrity, a BSD checksum byte of the **uncompressed** data is prepended to the **beginning** of the compressed data packet. This byte is for verification purposes only and **should not** be read as part of the compressed data.

Compressed output is in 16-bit words. Therefore, LZW compression has the potential to dramatically increase the size of packets too small or entropic to be efficiently compressed.

Formal Format Specification

A more detailed description of the format of a compressed LZW data packet follows:

- 0x00: BSD checksum byte.
- 0x01-*: The compressed payload.

Chapter V – Encryption

Denobo officially supports the establishment of a secure session between two socket-enabled agents. During a secure session, all packets transmitted over the socket are encrypted using an RC4-drop-4096 stream cipher. A shared secret key is agreed upon during handshaking and initialisation using a Diffie-Hellman key exchange. During this section, non-secret values are shown in blue and secret values in red.

The Key Exchange: An Overview

The precise details of the Diffie-Hellman key exchange used to establish a shared secret key between two peers without ever sending the key itself over the wire will not be discussed here. However, a brief overview of the key-exchange process in the official Denobo middleware will be outlined:

Stage 1: Private/Public Key Pair Generation

During construction, a socket-enabled agent or *SocketAgent* chooses a random large integer $Priv_L$ to act as its private key. From this, it calculates its public key Pub_L with the formula:

$$Pub_L = g^{Priv_L} \bmod p$$

Where *g* and *p* are respectively the generator and large prime given in the Appendix. [1]

Stage 2: Establishment of Shared Key

As part of its 100 (GREETINGS) packet, the local peer submits its public key Pub_L to the remote peer. If the remote peer wishes this session to be encrypted, it will respond with a 201 (BEGIN_SECURE) packet containing its public key Pub_R. The local peer now computes the shared secret key *K* with the formula.

$$K = {Pub_R}^{Priv_L} \bmod p$$

The remote peer does likewise with the local node's public key and its private key. A shared secret key has now been established.

At the moment, the shared secret is merely a large integer. To create a fixed array of bytes for use with the encryption algorithm, this large integer is converted to an ASCII string and hashed using the SHA-256 hash algorithm. The bytes contained in the resulting string are used as the key to the RC4-drop-4096 cipher.

Chapter VI – Message Structure and Serialisation

In the official Denobo agentware, messages consist of three parts:

- A 256-bit likely-to-be-globally-unique identifier, generated by hashing together several pseudorandom values. No definitive check for uniqueness is done however.
- A route, consisting of agent names, along which the message will travel to reach a named recipient.
- A payload, consisting of arbitrary data being relayed to the recipient.

Message Serialisation

Messages are serialised for transmission across sockets in query string format, containing each of the above data. For example:

id=2555bbf8ef7845e0143e290a714fa5ccfb62fea63590d403eff0d4cf1918f8dc&route=position%3d4%26route%3dAgent1%252cAgent2%252cSocketAgent1%252cSocketAgent2%252cRemoteDestination&data=Hello%20World%21

While the ID and payload fields are fairly evidently simply key-value pairs, the route field is a little more complex, consisting of a nested query string containing a serialised route.

Route Serialisation

A route attached to a Denobo message contains two parts:

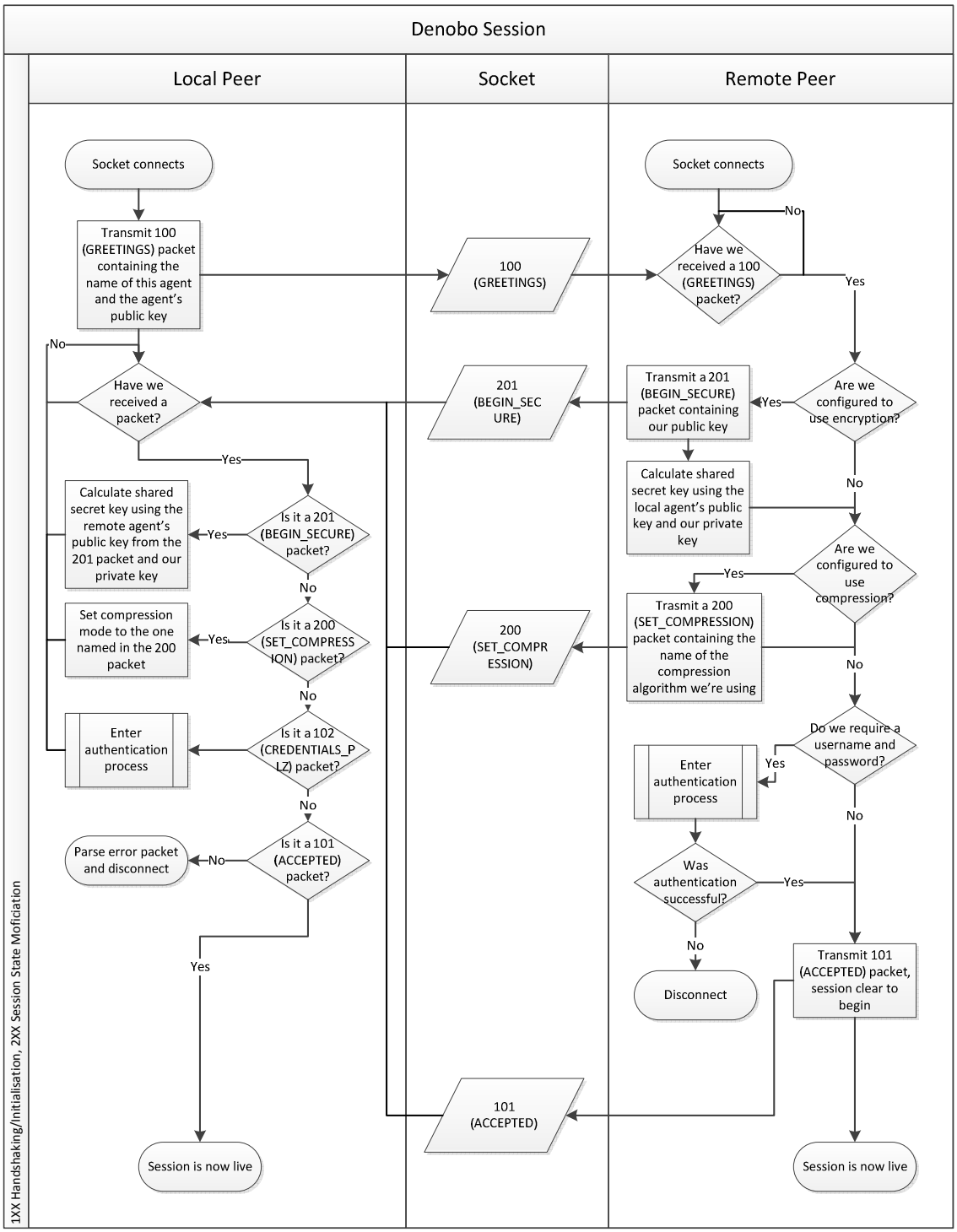
- A list of agent names, in sequence from the start of the route to the destination, with the message originator first and the recipient last.
- A position in the route. This is incremented every time the message passes through an agent on the way to its destination.

A route is serialised for transmission across sockets in query string format, containing each of the above data. For example:

position=4&route=Agent1%2cAgent2%2cSocketAgent1%2cSocketAgent2%2cRemoteDestination

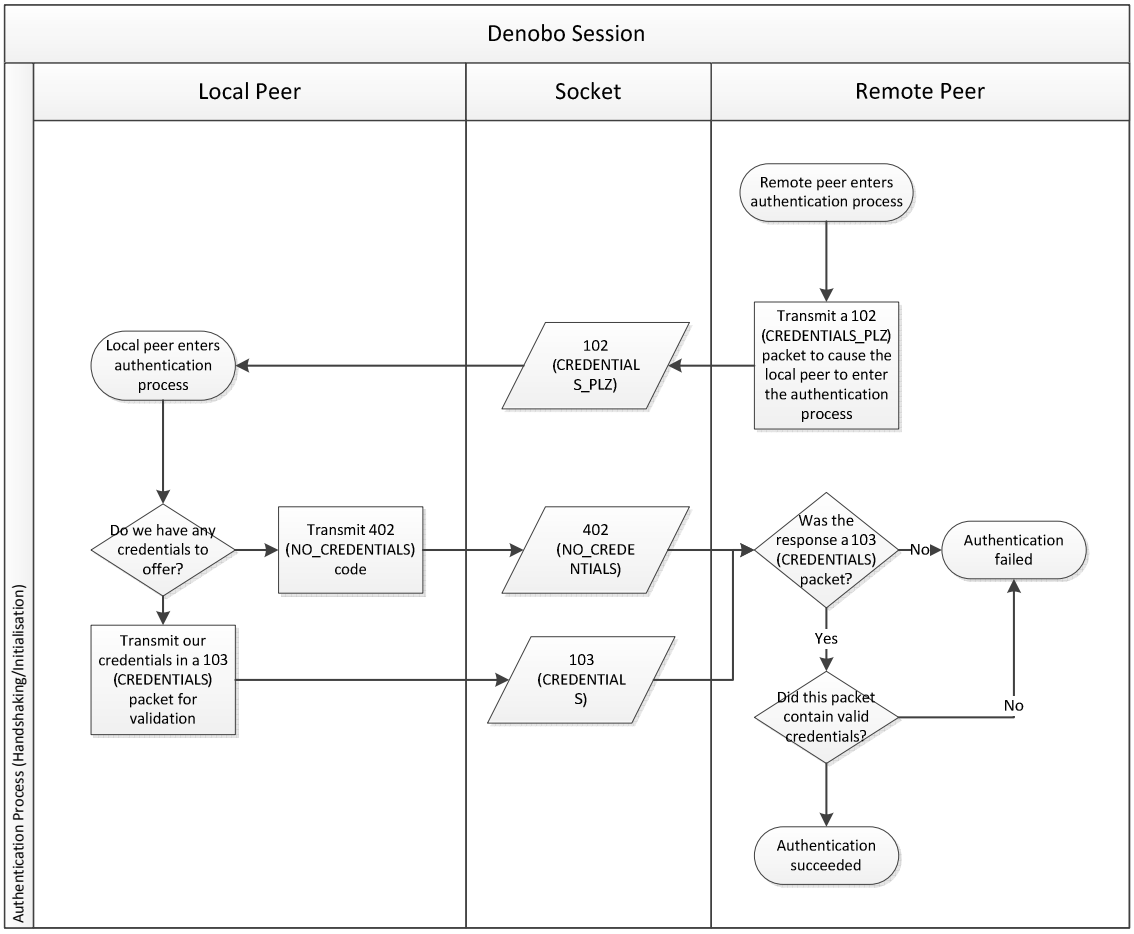
Chapter VII – Session Flow

Handshaking/Initialisation



Subprocess – Authentication

This diagram covers the subprocess shapes on the Handshaking/Initialisation diagram labelled “Enter authentication process”.



Defining a “Live” Session

A DMTP session is considered ‘live’ once 3XX packets containing messages and routing requests are clear to be transmitted. This is not diagrammed, as its highly asynchronous nature would make an illustration of limited use.

Appendix

[1] - 1536-bit MODP Group (Taken from RFC 3526)

The 1536 bit MODP group has been used for the implementations for quite a long time, but was not defined in RFC 2409 (IKE). Implementations have been using group 5 to designate this group, we standardize that practice here.

The prime is: 2^1536 - 2^1472 - 1 + 2^64 * { [2^1406 pi] + 741804 }

Its hexadecimal value is:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

The generator is: 2.