CS3099 micro:bit Network Protocol Definition

Supergroup B

April 2019

Version 4

Summary of Changes

+ Added configuration of *frequency band* being set for communications
+ Added *encrypted* bit to flags in header

- Reduced *packet ID* from 4 bits to 3 bits in flags byte
- Included notes discussing preventing impersonation using password in encrypted message
- Included notes on the meaning of the new encryption flag

4.1 Definition

4.1.1 Packet Structure

Each packet must be **at most** 128 bytes, and is split into a header and payload.

Header	Recipient ID	1 byte
	Sender ID	1 byte
	Message Type	1 byte
	Flags	1 byte
Payload	Content	124 bytes

Sender & Recipient IDs

Each of the 8 groups are assigned 4 IDs for their micro:bits. In the header, both the *Recipient ID* and *Sender ID* must come from the following table:

Name	IDs
Broadcast	000000
Group 1	000001, 000010, 000011, 000100
Group 2	000101, 000110, 000111, 001000
Group 3	001001, 001010, 001011, 001100
Group 4	001101, 001110, 001111, 010000
Group 5	010001, 010010, 010011, 010100
Group 6	010101, 010110, 010111, 011000
Group 7	011001, 011010, 011011, 011100
Group 8	011101, 011110, 011111, 100000

Message Type

The *Message Type* is 1 byte, and must be one of three three values below.

Request	00
Acknowledge	01
Data	10

Flags

The definition of the *Flags* byte is given as:

Priority	2 bits
Packet ID	3 bits
Encrypt	1 bit
Hop Count	2 bits

Note that the *priority* is most significant, while the hop count is least significant, in the byte.

Priority 2 bits, 00 is highest priority, 11 is lowest priority.

Priority	Value
High	00
Normal	01
Low	10
None	11

Packet ID The packet ID is a 3-bit number unique* to a message. It is only unique when considered alongside the sender ID - this allows each microbit to send and/or receive $2^3 = 8$ whole messages before an ID is reused.

A packet ID

Hop Count The hop count is the maximum number of times the packet should be forwarded, by other micro:bits, to reach the recipient before it is dropped.

The **maximum** is **3 hops** (flag value 11), and once it reaches 00, should not be forwarded any longer.

Payload

Acknowledgement Packet The *payload* of a packet with type *acknowledgement* is the 8-byte public key, followed by the 8-byte modulus n, which are used for encrypting the data payload.

Data Packet The *payload* of a packet with type *data* is **at most** 124 bytes, this **should** be an ASCII encoded string.

4.2 Encryption

Encrypt plaintext of block size 3 (bytes) into ciphertext of block size 4 (bytes).

4.2.1 Key Generation

1. Choose primes p and q, with $p \neq q$
2. Evaluate $n = pq$, with $n \in [2^{24},\ 2^{32} - 1]$
3. Evaluate $\lambda(n) = lcm(p-1,\ q-1)$, where lcm denotes *lowest common multiple*
4. Choose e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$, where gcd denotes the *greatest common divisor*
5. Evaluate d as the *modular multiplicative inverse* of e modulo $\lambda(n)$

- i.e. find d such that $e \times d = 1 \ \ (mod\ \lambda(n))$[1]

Then, the public key is e together with n, and the private key is d.

4.2.2 Encryption

General Algorithm

To encrypt an integer, m, using RSA, perform the following operation:

$$c \equiv m^e \ \ (mod\ n) \tag{4.1}$$

given public key e, and modulus n, we obtain ciphertext c.

Converting Byte Array to Integer

This implementation will encrypt 3-byte blocks of plaintext into a 4-byte block of ciphertext.[2] This is much more secure than simply encoding individual characters, whilst also avoiding the challenge of dealing with numbers which are hundreds of digits long and having to generate primes of similar size.

Example Consider the following example, where we wish to take the plaintext message `hello`, and convert it into blocks for encryption.

$$hello = [`h', \ `e', \ `l', \ `l', \ `o']$$

So let

$$bytes = [104, \ 101, \ 108, \ 108, \ 111]$$

For the first block, we take the first 3 bytes, $[104, \ 101, \ 108]$
We construct our block as follows:

$$block = (104 << 16) \mid (101 << 8) \mid (108 << 0) = 6841708$$

We then take this as our plaintext integer m which we encrypt using the modular exponentiation described previously.
The second block would be constructed as follows:

$$block = (108 << 16) \mid (111 << 8) \mid (0 << 0) = 7106304$$

Again, we encrypt this value.

Converting Integer to Byte Array

Consider the block value $block = 1764$. We convert into a little-endian byte array, using the following method:

$$bytes[i] \mid = (block >> (8 * i)) \ \& \ 255$$

Hence, this value converts into the following byte array:

$$bytes = [228, 6, 0, 0]$$

4.2.3 Decryption

General Algorithm

The corresponding decryption operation is as follows, now assuming we have a an integer, c:

$$c^d \equiv (m^e)^d \equiv m \ \ (mod\ n) \tag{4.2}$$

given private key d, and modulo n, we obtain plaintext m.

[1] *Modular Multiplicative Inverse* can be calculated using the *Extended Euclidean Algorithm*

[2] The message size increases due to the fact that ciphertext values lie in the range $[0, n-1]$, and n must be greater than the plaintext value. Hence, n must be *at least* $2^{3*8} = 2^{24}$.

4.3 Process

- All micro:bits who wish to use client-client communication **must** use group 86 on the communication channel used by the supergroup.
- All micro:bits should use *frequency band* zero.
- In this group, all micro:bits **should** be listening for requests to communicate.
- Any messages received which do not have the recipient ID matching a micro:bit's own ID **must** be forwarded.
  - The *hop count* should be read from the packet header
  - If the *hop count* is non-positive (0 or negative), the packet should be dropped;
  - If the *hop count* is greater than the *maximum hop count*, then set *hop count* to *maximum hop count*;
  - Otherwise, the hop count should be decremented and written back to the header;
  - The packet should then be sent over radio.
- A micro:bit which receives a *request* to communicate **should** send an *acknowledgement* packet back to the original sender containing their public key, with a unique packet ID. **Important**: the receiving micro:bit **must** only acknowledge a request **once** within a reasonable time frame - to avoid issues trying to repeatedly send a message.
- The sender should read the payload of the *acknowledgement* packet to get the public key which they use to encrypt the payload of the data packet.
- The sender then encrypts the payload of the *data* packet before sending - there is no requirement to acknowledge the receipt of the message.
- If the sender does not receive an acknowledgement from the recipient, the sender assumes the recipient is unavailable, and **should not** send the message.
- The recipient then decrypts the payload of the *data* packet using their private key.

4.4 Notes

- Packets with type *Request* **should not** have a payload.
- Timeout is defined as 2 seconds - this applies to waiting for acknowledgement for a request, and message from an acknowledgement.
- A packet with the same unique ID (Sender ID + Packet ID) as a recent packet **may** be ignored within a reasonable time frame, to avoid handling the same packet multiple times.
- The meaning of the *encrypt* bit in the *flags* byte depends on the packet type.
  - *Request* – says whether the requester wants the data to be delivered in plaintext or encrypted format.
  - *Acknowledgement* – has no meaning: depends on whether keys are included in payload.
  - *Data* – indicates whether the payload is encrypted.
- If the data was requested to be **encrypted** but the **key was not given** in the acknowledgement packet, then the data packet **should not be encrypted** – it *may* also be implemented that the **data is not sent** since a secure option is not available.
- If the data was request to be **plaintext** but **key was given** in the acknowledgement, the data packet **should not be encrypted**, to respect the request packet.

Notes on Impersonation

The following notes summarise the discussion in supergroup meetings during Semester 2 regarding preventing impersonation of users.

Purpose

Here we summarise the discussions at a supergroup level about preventing impersonation when communicating with other users over the radio.

The issue we aim to prevent is a user pretending to be another, by responding to packets which aren't addressed to them.

We discussed many options for this but the focus on having a system which was scalable and peer-to-peer were the most important factors.

Potential Solutions

Verification with Passwords

Two parties agree, in person, or via other secure medium, on a key or password which they will include in their encrypted communications. Therefore, any message from a specific address which does not contain the agreed password/key may be assumed to be insecure or untrustworthy.

This solution works on the assumption that parties can be trusted at a human level. That is, their identity can be verified independent of the device. This is possible by virtue of the university's matriculation card system, combined with the published list of group membership for this project. Hence, it can be verified that someone belongs to the group they claim to be a part of – the person must have the corresponding matriculation card, which confirms their name and matriculation number, and this can be checked against the list of groups for the project, as published on Student Resources.

Central Authority

All users of the network who wish to be considered *trusted* must register with a central authority, proving their identity matches their claimed identity.

Then, upon receipt of a message, a user can check with the authority if this message is actually from the identity claimed in the header.

This would be done using some kind of fixed, hardware value, such as the serial number of the microbit, which would not be shared with other users.

Implemented Solution

Solution	Advantages	Disadvantages
Verification with Passwords	Minimal change to protocol required to implement No protocol overhead -	Very manual method Relies on external verification of identity Requires secure channel to agree on password
Central Authority	Able to guarantee by trusted third party No requirement to store information on each device, just ask authority	Requires centralised system in peer-to-peer network Restricts scalability as all devices must verify their identity with trusted authority

Table 1: Pros and Cons of Impersonation Prevention Solutions

The chosen method to implement was the *Verification with Passwords* method, as it requires the least modification to the protocol, hence is the least disruptive. Further, it was agreed that relying on a central authority is not an ideal feature of a scalable, peer-to-peer network.

Version 3

Summary of Changes

+ Added requirement for public key to be given as payload of *acknowledgement* packet
+ Increased maximum packet size from 64-bytes to 128-bytes
+ Increased payload size from 60 bytes to 124 bytes
- Removed 4-packet system of breaking up messages
- Removed packet count from request packet payload
- Removed *sequence* flag from flags byte.
- • Extended *packet ID* from 2 bits to 4 bits in flags byte

3.1 Definition

3.1.1 Packet Structure

Each packet must be **at most** 128 bytes, and is split into a header and payload.

Header	Recipient ID	1 byte
	Sender ID	1 byte
	Message Type	1 byte
	Flags	1 byte
Payload	Content	124 bytes

Sender & Recipient IDs

Each of the 8 groups are assigned 4 IDs for their micro:bits. In the header, both the *Recipient ID* and *Sender ID* must come from the following table:

Name	IDs
Broadcast	000000
Group 1	000001, 000010, 000011, 000100
Group 2	000101, 000110, 000111, 001000
Group 3	001001, 001010, 001011, 001100
Group 4	001101, 001110, 001111, 010000
Group 5	010001, 010010, 010011, 010100
Group 6	010101, 010110, 010111, 011000
Group 7	011001, 011010, 011011, 011100
Group 8	011101, 011110, 011111, 100000

Message Type

The *Message Type* is 1 byte, and must be one of three three values below.

Request	00
Acknowledge	01
Data	10

Flags

The definition of the *Flags* byte is given as:

Priority	2 bits
Packet ID	4 bits
Hop Count	2 bits

Priority 2 bits, 00 is highest priority, 11 is lowest priority.

Priority	Value
High	00
Normal	01
Low	10
None	11

Packet ID The packet ID is a 4-bit number unique to a message. It is only unique when considered alongside the sender ID - this allows each microbit to send and/or receive $2^4 = 16$ whole messages before an ID is reused.

Hop Count The hop count is the maximum number of times the packet should be forwarded, by other micro:bits, to reach the recipient before it is dropped.

The **maximum** is **3 hops** (flag value 11), and once it reaches 00, should not be forwarded any longer.

Payload

Acknowledgement Packet The *payload* of a packet with type *acknowledgement* is the 8-byte public key, followed by the 8-byte modulus n, which are used for encrypting the data payload.

Data Packet The *payload* of a packet with type *data* is **at most** 124 bytes, this **should** be an ASCII encoded string.

3.2 Encryption

Encrypt plaintext of block size 3 (bytes) into ciphertext of block size 4 (bytes).

3.2.1 Key Generation

1. Choose primes p and q, with $p \neq q$
2. Evaluate $n = pq$, with $n \in [2^{24},\ 2^{32} - 1]$
3. Evaluate $\lambda(n) = lcm(p-1,\ q-1)$, where *lcm* denotes *lowest common multiple*
4. Choose e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$, where *gcd* denotes the *greatest common divisor*
5. Evaluate d as the *modular multiplicative inverse* of e modulo $\lambda(n)$

- i.e. find d such that $e \times d = 1 \pmod{\lambda(n)}$[3]

Then, the public key is e together with n, and the private key is d.

3.2.2 Encryption

General Algorithm

To encrypt an integer, m, using RSA, perform the following operation:

$$c \equiv m^e \pmod{n} \tag{3.3}$$

given public key e, and modulus n, we obtain ciphertext c.

Converting Byte Array to Integer m

This implementation will encrypt 3-byte blocks of plaintext into a 4-byte block of ciphertext.[4] This is much more secure than simply encoding individual characters, whilst also avoiding the challenge of dealing with numbers which are hundreds of digits long and having to generate primes of similar size.

Consider the following example where we have a plaintext message `hi` that we wish convert into an integer for encryption.

We take blocks of size 3-bytes, starting from the left of the string. Since the length is less than 4, we pad the right side with 0s. Hence, the 3-byte block looks like the following, in binary,

$$01101000, 01101001, 00000000$$

We then take this value and consider it as a single 24-bit number (which fits in a `uint64_t`), so we have

$$m = 011010000110100100000000 \ (6842624)$$

3.2.3 Decryption

General Algorithm

The corresponding decryption operation is as follows, now assuming we have a an integer, c:

$$c^d \equiv (m^e)^d \equiv m \pmod{n} \tag{3.4}$$

given private key d, and modulo n, we obtain plaintext m.

3.3 Process

- All micro:bits who wish to use client-client communication **must** use group 86 on the communication channel used by the supergroup.
- In this group, all micro:bits **should** be listening for requests to communicate.
- Any messages received which do not have the recipient ID matching a micro:bit's own ID **must** be forwarded.
  - The *hop count* should be read from the packet header
  - If the *hop count* is non-positive (0 or negative), the packet should be dropped;
  - If the *hop count* is greater than the *maximum hop count*, then set *hop count* to *maximum hop count*;

[3] *Modular Multiplicative Inverse* can be calculated using the *Extended Euclidean Algorithm*

[4] The message size increases due to the fact that ciphertext values lie in the range $[0, n-1]$, and n must be greater than the plaintext value. Hence, n must be *at least* $2^{3*8} = 2^{24}$.

  - Otherwise, the hop count should be decremented and written back to the header;
  - The packet should then be sent over radio.

- A micro:bit which receives a *request* to communicate **should** send an *acknowledgement* packet back to the original sender containing their public key, with a unique packet ID. **Important**: the receiving micro:bit **must** only acknowledge a request **once** within a reasonable time frame - to avoid issues trying to repeatedly send a message.
- The sender should read the payload of the *acknowledgement* packet to get the public key which they use to encrypt the payload of the data packet.
- The sender then encrypts the payload of the *data* packet before sending - there is no requirement to acknowledge the receipt of the message.
- If the sender does not receive an acknowledgement from the recipient, the sender assumes the recipient is unavailable, and **should not** send the message.
- The recipient then decrypts the payload of the *data* packet using their private key.

3.4 Notes

- Packets with type *Request* **should not** have a payload.
- Timeout is defined as 2 seconds - this applies to waiting for acknowledgement for a request, and message from an acknowledgement.
- A packet with the same unique ID (Sender ID + Packet ID) as a recent packet **may** be ignored within a reasonable time frame, to avoid handling the same packet multiple times.

Notes on RSA

The following notes summarise the research carried out by group B-8.

Operation

Key Generation

1. Choose primes p and q, with $p \neq q$
2. Evaluate $n = pq$
3. Evaluate $\lambda(n) = lcm(p-1,\ q-1)$, where lcm denotes *lowest common multiple*
4. Choose e such that $1 < e < \lambda(n)$ and $gcd(e, \lambda(n)) = 1$, where gcd denotes the *greatest common divisor*
5. Evaluate d as the *modular multiplicative inverse* of e modulo $\lambda(n)$
   - i.e. find d such that $e \times d = 1\ \ (mod\ \lambda(n))$
   - This can be done using the *Extended Euclidean Algorithm*

Then, the public key is e together with n, and the private key is d.

Encryption

The simplest form of RSA encryption is given below, and provides a method for encrypting a message - an integer - m.

$$c \equiv m^e\ \ (mod\ n) \tag{5}$$

given public key e, and modulus n, we obtain ciphertext c.

Decryption

The corresponding decryption algorithm is as follows, now assuming we have ciphertext - an integer - c.

$$c^d \equiv (m^e)^d \equiv m\ \ (mod\ n) \tag{6}$$

given ciphertext c, private key d, and modulo n, we obtain plaintext m.

Implementation Notes

Possible Encryption Solutions

Bytewise Encryption A possible implementation is to perform encryption and decryption in a bytewise manner. That is, to encrypt each character individually. However, there are flaws with this solution.

Assuming we take the unsigned numerical value of each byte (0 - 255), there is no guarantee that the ciphertext c will not overflow a single byte, for any given $m \in [0,\ 255]$.

- One option is to assert that $n \leq 256$, in which case, $c \in [0,\ 255]$, by definition.
- However, for encryption and decryption to work correctly, we **require m $<$ n**, hence we cannot encrypt the whole range of byte values $[0,\ 255]$ for any $n < 256$.

So, for bytewise encryption of all possible values $[0,\ 255]$, we require $n = 256$. That is, $p,\ q$ primes such that $n = pq$. There are **no such solutions**.

An alternative, and *recommended*, method is to only allow plaintext to occupy the values $[0,\ 127]$, the (non-extended) ASCII character set. In this case, with a value $n = pq \in [128,\ 255]$, the encryption **will work** correctly, and there are sufficient prime pairs which satisfy this to provide a *reasonable* level of security – at least as a demonstration of the method, given the limitations of this project.[5]

Block Encryption One issue with bytewise encryption is the strength, or lack thereof, the security of the method.[6] Each byte being encrypted individually means that each character has the same corresponding ciphertext character, and hence is easily cracked.

To improve the security of our encryption, consider taking the message in blocks of an arbitrary number of bytes long. In this case, consider blocks of 8 bytes – 8 bytes is a convenient block size as this is the size of a `long long`, or `uint64_t`. Then, take every group of 8 characters (bytes) to be the value of the plaintext, and encrypt this value using the method described earlier.

Assuming we maintain the restriction of plaintext values occupying the range $[0,\ 127]$, each block of 8 characters will take at most 63 bytes, and then the ciphertext will required at most 64 bytes, and is therefore contained within an 8-byte-block.

In this case, primes must be chosen such that $n = pq \in [2^{63},\ 2^{64} - 1]$.

Considerations for Current Protocol

Removing Packet Switching Given the fact that we do not yet have an procedure for handling the re-sending of lost packets, it might be more worthwhile – by that I mean it might increase the likelihood that we have a working system by the demonstration session – to drop the chunking/packet implementation in favour of using larger, single packets.

Increasing Packet Size Unless it proves to be significantly detrimental to the reliability, or range, of the radio communication, it could be worth increasing the packet size from the current 64 bytes to 256 bytes (or some other agreed size), hence accounting for the removal of our maximum-4-packet protocol discussed in the previous paragraph.

Key Exchange In order for a message to be sent in an encrypted state, the sender needs to know the public key of the recipient they are sending to. A possible solution to this involves extending the protocol to give an acknowledgement packet a payload.

When a request is made to a microbit, an acknowledgement is sent back, with the public key - comprised of e and n - is given in the payload of the packet. The sender then takes this key and uses it to encrypt the payload of their message packet, before they send to the recipient.

This avoids the issue of having to maintain a central party which stores public keys of all microbits, and ensures keys are always up to date, in the event that a public key for a given microbit has changed for some reason.

[5]This restriction to non-extended ASCII was proposed by group 1, and appears to solve the issues discussed in other methods considered here.

[6]This issue was highlighted by group 1, and the block encryption was proposed as a possible solution.

Version 2

Summary of Changes

- Re-purposed *checksum* flag as new flags *packet ID* and *hop count*

+ Added requirement to forward messages not intended for self

+ Added maximum hop count

- Removed requirement to ignore messages not intended for self

2.1 Definition

2.1.1 Packet Structure

Each packet must be **at most** 64 bytes, and is split into a header and payload.

Header	Recipient ID	1 byte
	Sender ID	1 byte
	Message Type	1 byte
	Flags	1 byte
Payload	Content	60 bytes

Sender & Recipient IDs

Each of the 8 groups are assigned 4 IDs for their micro:bits. In the header, both the *Recipient ID* and *Sender ID* must come from the following table:

Name	IDs
Broadcast	000000
Group 1	000001, 000010, 000011, 000100
Group 2	000101, 000110, 000111, 001000
Group 3	001001, 001010, 001011, 001100
Group 4	001101, 001110, 001111, 010000
Group 5	010001, 010010, 010011, 010100
Group 6	010101, 010110, 010111, 011000
Group 7	011001, 011010, 011011, 011100
Group 8	011101, 011110, 011111, 100000

Message Type

The *Message Type* is 1 byte, and must be one of three three values below.

Request	00
Acknowledge	01
Data	10

Flags

The definition of the *Flags* byte is given as:

Priority	2 bits
Sequence	2 bits
Packet ID	2 bits
Hop Count	2 bits

Priority 2 bits, 00 is highest priority, 11 is lowest priority.

Priority	Value
High	00
Normal	01
Low	10
None	11

Sequence The sequence number of the current packet, 00 being 1st packet, 11 being 4th packet.

Packet ID The packet ID is a pair of bits unique to a message. It is only unique when considered alongside the sender ID - this allows each microbit to send and/or receive 4 whole messages before an ID is reused.

Hop Count The hop count is the maximum number of times the packet should be forwarded, by other micro:bits, to reach the recipient before it is dropped.

The **maximum** is **3 hops** (flag value 11), and once it reaches 00, should not be forwarded any longer.

Payload

Request Packet The *payload* of a packet with type *request* is a single byte which indicates the number of packets in the message (maximum of 4).

Data Packet The *payload* of a packet with type *data* is **at most** 60 bytes, this **should** be an ASCII encoded string.

2.2 Process

- All micro:bits who wish to use client-client communication **must** use group 86 on the communication channel used by the supergroup.
- In this group, all micro:bits **should** be listening for requests to communicate.
- Any messages received which do not have the recipient ID matching a micro:bit's own ID **must** be forwarded.
  - The *hop count* should be read from the packet header
  - If the *hop count* is non-positive (0 or negative), the packet should be dropped;
  - If the *hop count* is greater than the *maximum hop count*, then set *hop count* to *maximum hop count*;
  - Otherwise, the hop count should be decremented and written back to the header;
  - The packet should then be sent over radio.
- A micro:bit which receives a *request* to communicate **should** send an *acknowledgement* packet back to the original sender, with a unique packet ID. **Important**: the receiving micro:bit **must** only acknowledge a request **once** within a reasonable time frame - to avoid issues trying to repeatedly send a message.

- The recipient should read the payload of the *request* packet to determine how many packets it expects the message to be.
- Once the recipient has acknowledged the message request, the sender **should** immediately send the whole message - there is no requirement to acknowledge the receipt of the message.
- If the sender does not receive an acknowledgement from the recipient, the sender assumes the recipient is unavailable, and **should not** send the message.

2.3 Notes

- Packets with type *Acknowledge* **should not** have a payload.
- There is no protocol-defined terminating character on the message, suggested that teams use length of message.
- Timeout is defined as 2 seconds - this applies to waiting for acknowledgement for a request, and message from an acknowledgement.
- A packet with the same unique ID (Sender ID + Packet ID) as a recent packet **may** be ignored within a reasonable time frame, to avoid handling the same packet multiple times.

Version 1

1.1 Definition

1.1.1 Packet Structure

Each packet must be **at most** 64 bytes, and is split into a header and payload.

Header	Recipient ID	1 byte
	Sender ID	1 byte
	Message Type	1 byte
	Flags	1 byte
Payload	Content	60 bytes

Sender & Recipient IDs

Each of the 8 groups are assigned 4 IDs for their micro:bits. In the header, both the *Recipient ID* and *Sender ID* must come from the following table:

Name	IDs
Broadcast	000000
Group 1	000001, 000010, 000011, 000100
Group 2	000101, 000110, 000111, 001000
Group 3	001001, 001010, 001011, 001100
Group 4	001101, 001110, 001111, 010000
Group 5	010001, 010010, 010011, 010100
Group 6	010101, 010110, 010111, 011000
Group 7	011001, 011010, 011011, 011100
Group 8	011101, 011110, 011111, 100000

Message Type

The *Message Type* is 1 byte, and must be one of three three values below.

Request	00
Acknowledge	01
Data	10

Flags

The definition of the *Flags* byte is given as:

Priority	2 bits
Sequence	2 bits
Checksum	4 bits

Priority 2 bits, 00 is highest priority, 11 is lowest priority.

Priority	Value
High	00
Normal	01
Low	10
None	11

Sequence The sequence number of the current packet, 00 being 1st packet, 11 being 4th packet.

Checksum The checksum is evaluated on the contents of the payload only, and is defined as follows:

the ith bit of the checksum is equal to the sum of every nth bit in the payload, where $n \% 4 = i$

Payload

Request Packet The *payload* of a packet with type *request* is a single byte which indicates the number of packets in the message (maximum of 4).

Data Packet The *payload* of a packet with type *data* is **at most** 60 bytes, this **should** be an ASCII encoded string.

1.2 Process

- All micro:bits who wish to use client-client communication **must** use group 86 on the communication channel used by the supergroup.
- In this group, all micro:bits **should** be listening for requests to communicate.
- Any messages received which do not have the recipient ID matching a micro:bit's own ID **must** be ignored.
- A micro:bit which receives a *request* to communicate **should** send an *acknowledgement* packet back to the original sender.
- The recipient should read the payload of the *request* packet to determine how many packets it expects the message to be.
- Once the recipient has acknowledged the message request, the sender **should** immediately send the whole message - there is no requirement to acknowledge the receipt of the message.
- If the sender does not receive an acknowledgement from the recipient, the sender assumes the recipient is unavailable, and **should not** send the message.

1.3 Notes

- Packets with type *Acknowledge* **should not** have a payload.
- Packets with no payload should have a checksum of 0.
- There is no protocol-defined terminating character on the message, suggested that teams use length of message.
- Timeout is defined as 2 seconds - this applies to waiting for acknowledgement for a request, and message from an acknowledgement.