Module 8.1 – Data Serialization

Required reading material:

- [1] Brian "Beej Jorgensen" Hall, "Beej's Guide to Network Programming, v3.1.11". April 2023. https://beej.us/guide/bgnet/html/split/
    - Chapter 7.5 – 7.6

In this module we'll briefly cover the topic of data serialization in socket programming.

In module 2.3 we mentioned how different computer architectures represent data in different ways, most significant vs least significant bits, memory layout and representation, etc. In this module we'll address the topic more formally and we'll go over a few approaches used to handle data transfer across networks.

Network Byte Ordering vs Host Byte Ordering and why it is important.

Let's cover the basics of data transmission across different machines and how data is represented as it crosses the wire.

If we think back to an Assembly programming course, data organization, or Operating System course, we'll remember that different processors and computer architectures represent bits and bytes in different ways. There are two common ways to write bytes on a computer system, Big-Endian and Little-Endian, both of which refer to whether the least or most significant byte is the trailing byte in a message.

We'll use the two-byte hexadecimal string: 0xb34f as our example string.

Big-Endian (also known as Network Byte Order): this is the commonly agreed upon byte ordering where bytes are represented in the sequence the way you'd expect them to. That is, the byte string 0xB34F is represented as 0xB3, 0x4F on a Big-Endian system.

Little-Endian (known as Host Byte Order) on the other hand, stores the bytes in reversed order, that is the string 0xB34F would be stored in memory as the sequential bytes 0x4F followed by 0xB3. There are computer systems and processors which use this data representation method.

Also keep in mind that numbers are represented in two different ways: short (two bytes) and long (four bytes) and they need to be converted or handled appropriately whenever you send data over a network.

As we can expect, having two different data representation methods poses a problem when transmitting data over a network. To overcome this issue, the socket API provides a set of functions to convert between the different data representation methods. The big point we're trying to make here is that you'll want to convert your data to Network Byte Order as you send out over the network and likely convert it to Host Byte Order when you receive it on the endpoint (depending on the processor: Intel vs other processors, etc.). This might not be a big issue in this course since we'll almost always be running our server and client applications on the same system and so we'll not write our code with this mind. This issue becomes a real concern once you start transmitting data across different networks, such as the

internet, where data passes through multiple devices where they might use different processors with different architectures.

What's the best strategy when writing code that needs to account for network vs host byte ordering?

- You just get to assume the Host Byte Order isn't right, and you always run the value through a function to set it to Network Byte Order. The function will do the magic conversion if it has to, and this way your code is portable to machines of differing endianness.

- Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire and convert them to Host Byte Order as they come in off the wire.

Function	Description
`htons()`	host to network short
`htonl()`	host to network long
`ntohs()`	network to host short
`ntohl()`	network to host long

Data Serialization

Read "Beej's Guide to Network Programming, chapter 7.5 – 7.6.

While sharing text (ascii) is usually fine and not that big of an issue, exchanging objects, structures, and other type of data types in a program becomes a lot trickier when we try to send that data over the network as text.

Let's use the following C structure as an example:

```C
struct car
{
    char *model;
    char *vin;
    int max_speed;
}
```

If we attempt to send it over the network as such, it won't work as we expect it.

```
send(my_car*)
```

We'll run into issues:

- The function expects a buffer
- Different compilers will pack the structure differently
- You will likely need to do something along these lines

```
//function is missing the rest of the parameters for brevity
•  send(my_car->model)
•  send(my_car->vin)
•  send(my_car->max_speed)
```

and the client will have to do something along these lines:

```
//client
recv(my_car)
//unpack the data somehow, etc.

or know the order it is sent (will always have to be the same)

recv(my_car->model)
recv(my_car->vin)
recv(my_car->max_speed)
```

As you can easily tell, it gets really complicated really fast. To address these and many other issues, data serialization and de-serialization protocols and libraries have been designed. In the real world, we as developers would likely utilize one of the readily available libraries and not have to create our own.

As explained in the reading material, the most common approach to share data across networked systems is for all systems involved to agree on some protocol or data scheme when sending and receiving data. A few of those approaches are 1) to create your own marshalling/de-marshalling protocols, 2) using XML, JSON, or other standardized data transmission protocol.

For this module we'll use a combination of resources in different languages, C, C#, and Python. We won't be implementing our own serialization schemes but rather will be learning about the standard libraries used in the real world.

XML

https://en.wikibooks.org/wiki/XML_-_Managing_Data_Exchange/Introduction_to_XML

https://learn.microsoft.com/en-us/dynamics365/business-central/across-how-to-use-xml-schemas-to-prepare-data-exchange-definitions

https://aws.amazon.com/what-is/xml/

https://aws.amazon.com/compare/the-difference-between-json-xml/

https://learn.microsoft.com/en-us/dotnet/standard/serialization/

https://realpython.com/python-serialize-data/

JSON

https://learn.microsoft.com/en-us/dotnet/standard/serialization/

https://aws.amazon.com/compare/the-difference-between-json-xml/

https://realpython.com/python-serialize-data/

https://github.com/json-c/json-c

https://github.com/DaveGamble/cJSON

Protocol Buffers – protobuf

Another popular library is protocol buffers or protobuf, which can be found here: https://protobuf.dev/

This library supports multiple languages and takes a similar approach to “Beej's Guide to Network Programming, chapter 7.5 – 7.6.