Network Message Transmission

Network Message Transmission

- A network connection only knows about sending binary data (e.g., Python "bytes" objects).
- Any interpretation of that data must be done explicitly by the end points, i.e., applications. Sometimes special processing is needed to ensure proper interpretation. Example message types:
  - Binary data (e.g., binary file such as jpeg, mpeg, etc). This is not a problem for the network connection. Everything works transparently.
  - Text: e.g., textual input, text files. These need to be encoded into binary at each end point. We will discuss this case later.
  - Data objects (e.g., integers, floating point numbers, C structs, etc. Let's consider this case.

Sending Data Objects Over the Network

Example: What if multi-byte objects are sent over a network connection? Can problems arise?

Byte Order and Alignment

Byte Order (Endianness)

Machine Formats and Endianness

- Suppose a machine needs to store a 16-bit (two byte) integer.
- It may be stored in memory either least-significant-byte-first, or, most-significant-byte-first. The option used depends on the machine architecture. Some processors do it one way, some the other (although there are some that can use both options).
- little-endian machine: The least significant byte appears first, i.e., at the lower address. It is followed by the most significant byte (e.g., Intel x86 and x86-64).
- big-endian machine: The most significant byte appears first, i.e., at the lower address. It is followed by the least significant byte (e.g., Motorola 68000 series)

  (Note ARM Version 3 and above is bi-endian.)

Endianness

- Why do we care?
  - When machines of different endianness communicate certain data, these differences must be taken into account.
- By convention, when data is communicated that must be interpretted as certain multi-byte objects, they are sent in big-endian format.

  <u>Big-endian is therefore also referred to as network byte order.</u>
- A little-endian machine will store this data locally in little-endian format. The data will have to be converted to big-endian when it is transmitted or used on the Internet.

ntohl, ntohs, htonl, htons

- Socket libraries have functions to convert basic data types between host and network byte order. In Python, for example, we have
- **`socket.htonl(x)`**
  - Convert 32-bit positive integer (x) from host to network byte order ("host-to-network-long").
- **`socket.htons(x)`**
  - Convert 16-bit positive integer (x) from network to host byte order ("host-to-network-short).

ntohl, ntohs, htonl, htons

- Socket libraries have functions to convert basic data types between host and network byte order. In Python, for example, we have
- **`socket.ntohl(x)`**
  - Convert 32-bit positive integer (x) from network to host byte order ("network-to-host-long").
- **`socket.ntohs(x)`**
  - Convert 16-bit positive integer (x) from network to host byte order ("network-to-host-short).
- See byte_order_conversion_examples.py

Sending Data Over Networks

- Let's do some examples that show how this can be done.
- First we will try sending some different size integers over a TCP connection. The receiver is written in C, the sender in Python.
- Send/Receive ints directly:
  - Receiver: TCP_receive_int.c (ReadIntFromClient.c)
  - Sender: TCP_send_int.py
- Send/Receive ints network byte order:
  - Receiver: TCP_receive_int_nbo.c (ReadIntFromClientNBO.c)
  - Sender: TCP_send_int_nbo.py

Exchanging Messages with Multiple Fields

- Compilers typically align data structures to start on word boundaries, i.e., they pad fields with zeros, so that word boundaries are achieved.
- A 32-bit machine uses 4 byte words, a 64-bit machine uses 8 byte words
- Data structures will use a different number of bytes on different machines, depending upon compiler padding.
- To deal with this, message fields must be defined so that the are consistently aligned between different machines.
- Functions have been defined that will take care of this alignment.

struct_padding_alignment_example_memory.c

Data Alignment

Alignment Example

Assume we have a word size of 32 bits. Here we have four consecutive words in memory:

0x00000000				0x00000004				0x00000008				0x0000000C			

If we write a 4-byte int into these locations, here is what we get. This is properly aligned.

0x00000000				0x00000004				0x00000008				0x0000000C			
i	i	i	i												

Instead, assume that we write a 1-byte char, a 2-byte int and then a 4-byte int.

0x00000000				0x00000004				0x00000008				0x0000000C			
c	s	s	i	i	i	i									

The 4-byte int is now misaligned. On some machines it may require two memory accesses and bit shifting to extract it. This can be avoided with “padding”.

Alignment Padding

0x00000000				0x00000004				0x00000008				0x0000000C			
c		s	s	i	i	i	i								

In this case, one byte of padding is added after the char. This re-aligns access to the 4-byte int.

C Compiler Alignment

- C compilers automatically pad variables. This is explicitly defined in the standards to which they adhere.
- e.g., a C struct is always aligned to the largest type's alignment requirements.
- A general rule is that scalar variables cannot "span" memory addresses, e.g., an 8-byte variable must begin at an address that is divisible by 8, 4 by 4, 2 by 2, etc.
- This ensures that an array of scalar variables will be aligned properly.

C Compiler Alignment

- chars can start on any byte address, but 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. Signed or unsigned makes no difference.

Exchanging Messages with Multiple Fields

- The multiple fields of a message could have fixed or variable length.
- Consider the case where we would like to exchange a C struct over a network connection.
- In this case we need to worry about data alignment.
- See padding/alignment example:

`struct_padding_alignment_example_memory.c`

Sending C Structs Over Networks

- **Avoid this if at all possible!**
- There are various serious portability issues to deal with, e.g.,
  - Endianness of individual struct members
  - Different struct padding
  - Different byte sizes of the underlying member types.
  - Some types make no sense after transmission, e.g., pointers.
- The basic approach would be to "serialize" the struct in a known way, then reverse the operation at the receiving end.

Python struct Module

- struct does conversions between Python values and C structs, input as Python bytes objects.
- This module can be used for communicating standard C structs over network connections.
- Format strings are used to describe the layout of the C structs and the Python value conversions
- See https://docs.python.org/3/library/struct.html

Python struct Module

- i.e., the Python struct module:

  import struct

```
struct.pack('@hiq', si, i, lli)
struct.unpack('@hiq', si, i, lli)
```

- The format code is interpretted as follows:

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network (BE)	standard	none

- h (short int), i (int), q (long long int)
- Padding/alignment only occurs for native option!

struct_module_examples.py

Code Examples

- Send/Receive ints directly:
  - Receiver: TCP_receive_int.c (ReadIntFromClient.c)
  - Sender: TCP_send_int.py
- Send/Receive ints network byte order:
  - Receiver: TCP_receive_int_nbo.c (ReadIntFromClientNBO.c)
  - Sender: TCP_send_int_nbo.py
- Send/Receive struct directly:
  - Receiver: TCP_receive_struct.c (ReadStructFromClient.c)
  - Sender: TCP_send_struct.py

Code Examples

- Send/Receive struct with network byte order
  - TCP_send_recv_struct_nbo.py

Can we send/exchange more complex data types over a network connection?

Data Serialization

Data Serialization

- Serialization is the converting of data objects into a form that can be transmitted over a network or stored (e.g., in a file), and then reconstructed into the same language objects later (also similar to "marshalling")
- Common serialization formats:
  - Extensible markup language (XML)
  - JavaScript Object Notation (JSON)
  - Yet Another Markup Language (YAML)

JSON

- is simple and human readable
- The JSON format is commonly used for serializing and transmitting structured data over network connections.

  e.g., when data needs to be transmitted between clients and servers in web applications.
- Some web services and APIs use the JSON format to distribute public data.
- used for configuration definition

JSON

- language-independent data format
- Originally came from JavaScript, but many programming languages include libraries to generate and parse JSON-format data.
- JSON's basic data types are string, boolean, array, objects (i.e., associative arrays, dictionaries or hashes), null.
- JSON is built on two structures:
  - a collection of name/value pairs, e.g., Python dictionary
  - an ordered list of values, e.g., Python list
- Therefore, JSON strings start with "[" or "{". Whitespace is ignored between language elements

JSON Examples

```
{
  "red"     : "#f00",
  "green"   : "#0f0",
  "blue"    : "#00f",
  "cyan"    : "#0ff",
  "magenta" : "#f0f",
  "yellow"  : "#ff0",
  "black"   : "#000"
}
```

JSON Examples

```
{
  "cars" : {
            "Nissan" : {
              "Sentra": {"doors":4,"transmission":"automatic"},
              "Maxima": {"doors":4, "transmission":"automatic"}
              },
            "Ford" : {
              "Taurus": {"doors":4,"transmission":"automatic"},
              "Escort": {"doors":4, "transmission":"automatic"}
              }
            }
}
```

Python json Module Basic Usage

- import json
- json.dumps(object, <option arguments>)
  - serialize the object into a JSON string
- json.loads(s, <option arguments>)
  - deserialize s, which can be string, bytes or bytearray object.
- json_colors_example.py
- json_readfile_example.py
- TCP_send_recv_json.py