1

Good evening. I’m Taka. I'd like to talk a little about **MessagePack**.

2

Before I do that though, let me **introduce myself**.

I'm from TOKYO, JAPAN.

I'm developing **publish/subscribe style** IoT, Internet of Things platform, based on **websockets** using **MessagePack**.

I'm a **committer** on the msgpack-c open source project.

msgpack-c is a **C and C++** implementation of the **MessagePack specification**.

The C++ version is a **native** implementation. **Not** just a **wrapper** on the C version.

I have some other related experience.

I wrote the Boost Meta State Machine Guide.

Let's move on to MessagePack.

3

MessagePack is a **binary** data format. But It's like **JSON**.

<u>This</u> is an example of some JSON data and <u>this</u> is the **equivalent** MessagePack data.

MessagePack is used by **redis, fluentd, Facebook Messenger**, and others.

4

Which MessagePack **features** are the **same** as **JSON**?

Both are **portable**, both contain **basic type information**.

They are a **composite** data structure.

5

A **composite** data structure means that **arrays** and **maps** can **contain** msgpack objects.

The **differences** between **JSON** and **MessagePack** are ...

MessagePack is **more compact** than JSON.

MessagePack is **binary coded**. So it can **handle** binary data **without text encoding** such as Base64.

MessagePack is **easier to parse**, it requires **less computing power**.

6

You may be interested in the **difference** between **MessagePack** and **Boost Serialization**.
The **goals** of **MessagePack** and **Boost Serialization** are different.
MessagePack's goal is **interexchange** data with **different** programming languages.
Boost serialization's goal is **serializing** and **deserializing** every C++ data **including their relations**.
For example, <u>these three shared pointers</u> point to the **objects** that have **value 42**, but the **first two shared pointers** are **sharing** the **same** object.
Boost serialization can **serialize** and **deserialize** these kinds of **relations**.
MessagePack can be used as a **portable binary archive** of Boost Serialization.
You can get the code of **msgpack archive** for **Boost Serialization** from my **github**.
The original version is written by **Norihisa**.

7

Here is a **list** of **programming languages** that support **MessagePack**.

8

Let's look at a **code example**.
The **client** needs to include <u>msgpack.hpp</u>. A C++ version of **Msgpack** is a **header only** library.

<u>The top part of the code example</u> demonstrates how to pack a tuple.
We can generate a byte stream that is MessagePack format using the 'pack‘ function.
The first argument of the pack function is a stream.
A stream is an object of any type that has a write member function.

<u>The middle part of the code example</u> demonstrates how to **unpack**.
We can **generate** a **MessagePack object** from a byte stream using the **‘<u>unpack</u>’** function.
A <u>MessagePack object</u> is a simple **variant** type. It is implemented **using a union**.
Although we can use a MessagePack object **directly**, it is usually more useful **to convert to C++ types**.
<u>The last part of the code example</u> demonstrates how to convert **from msgpack objects *to C++ types***.
Using the **‘<u>as</u>’** member function template, we can **convert** to **any C++ types** that is **adapted** MessagePack.

9

MessagePack also **provides** a **stream deserializer** named **unpacker**.

It has **four** member functions. reserve_buffer, buffer, buffer_consumed, and next.

Let's look at some client code.

10-15

These four member functions are used in the client code.

This code is a part of the **packet receiving function**.

Let's say the client tries to **read 100 bytes at a time**.

This is a packet receiving loop.

When a packet is **arrived**, the client calls the **reserve_buffer** function.

The unpacker **prepares** buffer memory **internally**, similar to the std::vector's reserve.

And then, the client copies **the data it receives** to **the prepared buffer**. In order to access **the internal buffer,** the client calls the **buffer()** function.

Then, the client **notifies** the **unpacker** the **actual read size** using the **buffer_consumed()** function.

The inner while loop generates MessagePack objects that are **wrapped in the type** named **msgpak::unpacked**.

For **each msgpack format data** in the buffer, the **'next'** function returns **true** and sets the variable **result**.

If there is **no complete MessagePack format data** in the buffer, then the **'next'** function returns **false**.

Even if the function **'next'** returns **false**, the unpacker **preserves the context** so that the unpacker can **continue parsing** when the **'next'** function will **call again**.

16

Another characteristic of MessagePack C++ is **zero-copy deserialization.**

Here is an example of an **unpacker buffer** that contains **Msgpack format data**.

The data is **an array** that has **three elements**.

17

After unpacking the buffer, the client **accesses** the object named <u>**unpacked**</u>.

The <u>**unpacked**</u> contains an <u>**msgpack::object**</u>, and the object contains <u>**three sub objects**</u>.

The sub objects are **allocated on the memory pool** named **zone**.

When the type of object is **string** or **bin**, the objects <u>**refer to unpacker buffer**</u>.

Strictly speaking, you can choose <u>**reference or copy**</u>.

Let's say we choose a **reference**.

So far, **no copies** are occurred.

18

The **client** might **convert** from the **msgpack::object** to **C++ types.**

If you choose a reference type **such as boost string_ref**, **no copies are occurred**.

You can also choose a **copy** operation.

For example converting to a **std::vector.** The data is **copied** to the **vector**.

19

MessagePack provides **adaptors** for basic C++ **types** and **containers**.

Now, we just started to support boost types.

20

You can also **adapt** your **class** to **MessagePack** using <u>MSGPACK_DEFINE</u> macro.

The **base** classes can **adapt** using <u>MSGPACK_BASE</u> macro.

This is an **intrusive** approach.

MessagePack also provides a **non-intrusive** approach.

You can see that the **URL** on the **slide**.